Noah Gundotra Follow
UC Berkeley '21 Computer Science and Math Major
Jun 4 · 11 min read

Code2Pix—Deep Learning Compiler for Graphical User Interfaces

uizard × SUSA
Statistics Undergraduate Student Association

Hello! My name is Noah Gundotra, I'm a sophomore at UC Berkeley. I'm presenting original research produced by UC Berkeley's Statistics Undergraduate Student Association (SUSA) in collaboration with Uizard.

Within SUSA, I led a team of 6 other UC Berkeley undergraduates to conduct this project during the 2018 Spring semester. The following work is the culmination of many unseen hours of work, failure, and fun from our group. We hope you will enjoy reading about our research!

There are 6 sections to this blog post, with a short Acknowledgements section.

TL;DR—Abstract

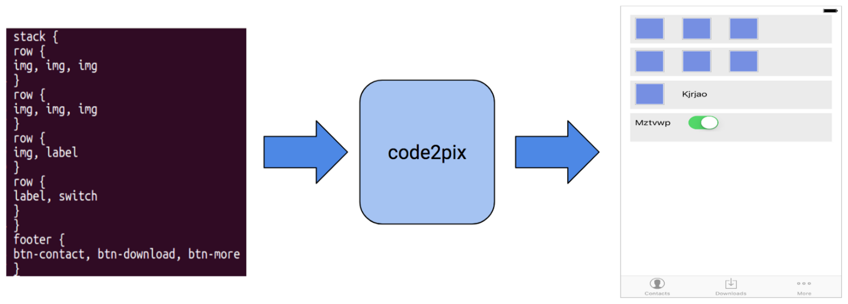
We provide a deep learning solution to the problem of generating Graphical User Interfaces (GUIs) from a textual description. This research could be used to improve the accuracy of Pix2Code, and future models which generate code from graphical user interfaces (the opposite task); this problem is referred to as User Interface Reverse Engineering in the Human-Computer Interaction literature.

Additionally, intermediary work hints at the possibility of a single textual to GUI compiler that works across multiple platforms.

1—Intro

When I first read the Pix2Code paper I was astounded. A single person, Tony Beltramelli, came up with a new application for deep learning, implemented an approach, and open-sourced it. The paper presented a deep learning model capable of turning sketches of graphical user interfaces into actual code. This seemed revolutionary to me. I think this was one of those things that I saw, and suddenly I understood why so many people were interested in machine learning—I really felt it was "artificial intelligence."

Pix2Code is a seriously awesome feat of machine learning engineering. To improve it, Tony suggested experimenting with a complementary project. Called Code2Pix, it would do the opposite of Pix2Code. Pix2Code goes from GUIs and turns them into textual descriptions in a Domain Specific Language (DSL). Code2Pix takes in textual descriptions in a DSL and generates GUIs. Thus Code2Pix is effectively a deep learning "compiler" able to render interfaces. To avoid confusion with true compilers, we will refer to this as a **deep learning renderer**.



```
stack {
row {
img, img, img
}
row {
img, img, img
}
row {
img, label
}
row {
label, switch
}
}
footer {
btn-contact, btn-download, btn-more
}
```

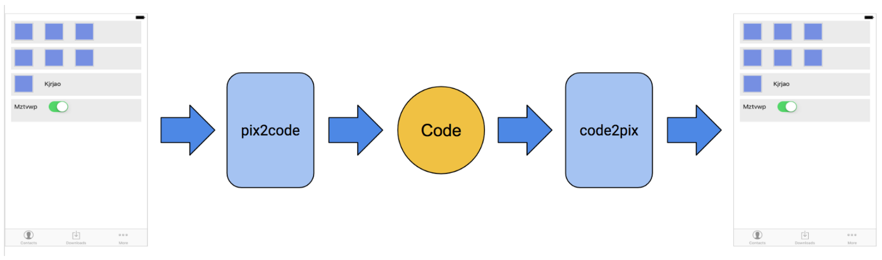
Simplified visual description of code2pix's inputs and outputs. This particular example is from the iOS dataset provided in the original pix2code repo.

In the spirit of Pix2Code, we are releasing all code associated with this project. We hope that we will inspire other engineers to work on this project and help the field move forward by contributing to the community.

2—Why Code2Pix?

We set out to improve Tony's model, Pix2Code by building a Generative Adversarial neural Network (GAN). Recent work in this field has shown that neural networks pitted against each other (GANs) can come up with better captions for images than standard models. This requires a ***generator*** model and a ***discriminator*** model to work. Pix2Code can be used to generate the code (DSL) for a user interface, and we can use Code2Pix to see how close the captions are at capturing the real image. Pix2Code would be the generator and Code2Pix would be the discriminator. Together, they could form a GAN which might be able to generate code for GUIs more accurately than Pix2Code by itself.

Code2Pix does the same work as any normal renderer. In fact, any renderer that handles user interface code is a "code to pix" system. However, Code2Pix is different because it is differentiable. This means that we can use Code2Pix, our deep learning renderer, to train other deep learning models by propagating error signals through its layers. This is not possible with a standard renderer.



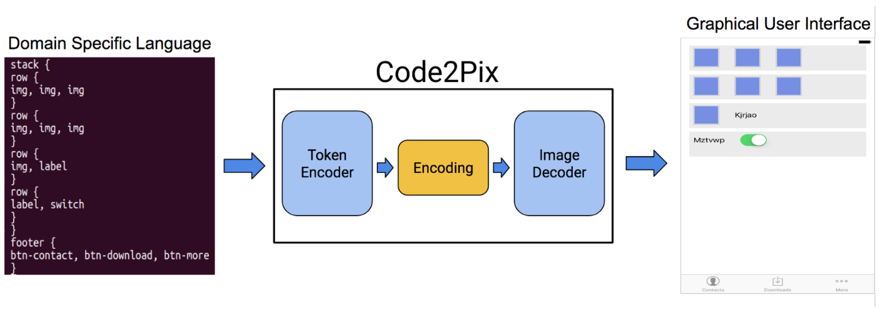
Pix2Code GAN architecture. This is an example of how Code2Pix could be used in conjunction with Pix2Code to improve Pix2Code accuracy in subsequent research.

3—Solving Code2Pix

This was the most fun and most humbling part of leading this project. Our original approach took us 2–3 months to build and iterate upon, but the final step ended up not using any of our previous work. The final working model turned out to be a very simple, almost naive approach. Even so, I believe our process is worth sharing, for educational purposes, as well as for demonstrating an interesting and relatively new application of using deep learning for cross platform user interface design. Moreover, we believe this project is a great example that simple and elegant solutions can very often perform better than complicated ones.

Approach

The problem of turning code into pictures can be broken down into three pieces. First, we needed to use a DSL to capture user interfaces to reduce the problem complexity and make it practical to use a deep learning system. Second, we would create an autoencoder for each dataset. Third, we would use the decoder portion of the autoencoder in the Code2Pix model to translate from the encoding space to pixels that render the GUI.



```
stack {
row {
img, img, img
}
row {
img, img, img
}
row {
img, label
}
row {
label, switch
}
}
footer {
btn-contact, btn-download, btn-more
}
```

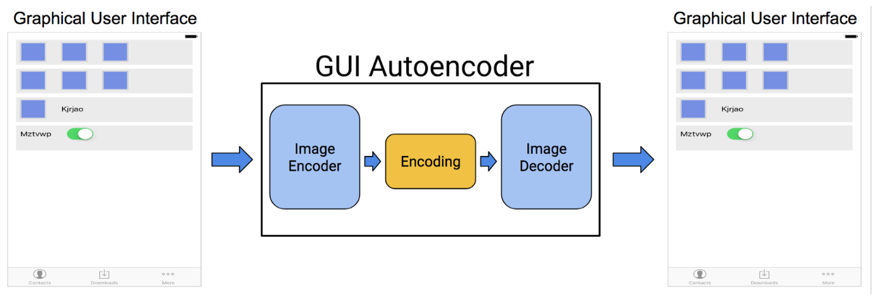
Our approach to building code2pix. Separate the model into two distinct portions. One for handling the textual information, and the other for constructing the GUI. We refer to these respectively as the "token encoder" and the "image decoder."

Lucky for us, Tony had already created a domain specific language (in this case the DSL can be seen as a pseudocode for user interfaces) that can be compiled to actual code for each dataset (i.e. native iOS, native Android, web HTML/CSS). Pix2Code and Code2Pix use the same domain specific language that capture the essential features of each dataset. This took care of our first step. If you'd like to see how this is done, take a look here.

Our next step is to train the autoencoders with the goal to use a portion of the autoencoder models to act as the "image decoder" in Code2Pix.

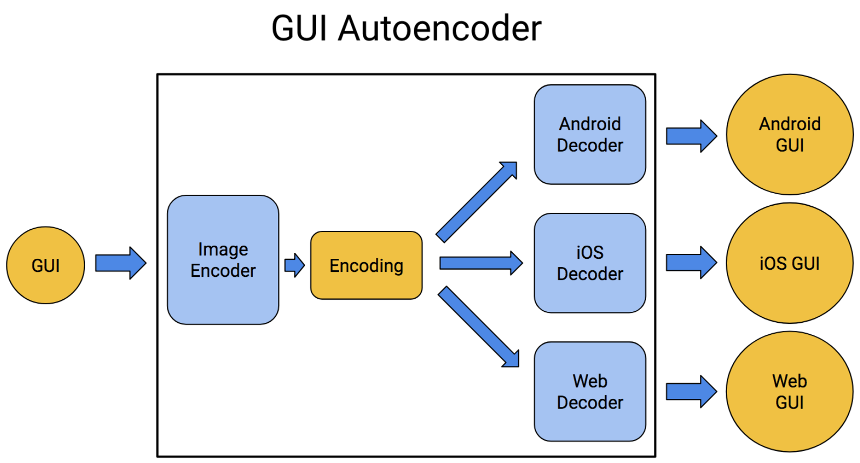
Autoencoders

These are special models in deep learning that are trained on a dataset as a precursor to solving a task on that data. Autoencoders learn to compress data to a lower dimensional encoding, then also learn to decompress the encodings to the original data. They are a great way to learn important features of the data. The encoder portion will often be imported into other deep learning architectures to transfer the features it learned to encode. This is known as **transfer learning**. The decoder portion is of particular interest to us because it learns to interpret abstract meaningful encodings and turn them into pictures of user interfaces.



If we train an autoencoder that can compress and decompress GUIs, then we get a free "image decoder" to use in code2pix.

We tried out standard autoencoder architectures to solve this task, but the decoders learned were not very useful in solving our task. So we designed a new architecture: a multi-headed model to solve the task simultaneously for all three datasets, Android, iOS, and Web. In this model, dubbed "Hydra", the decoder that interpret the image encoding is dependent upon the dataset the image came from. For example, to pass an Android image through the model, we would manually choose to pass it through the Android decoder head to get our final result.

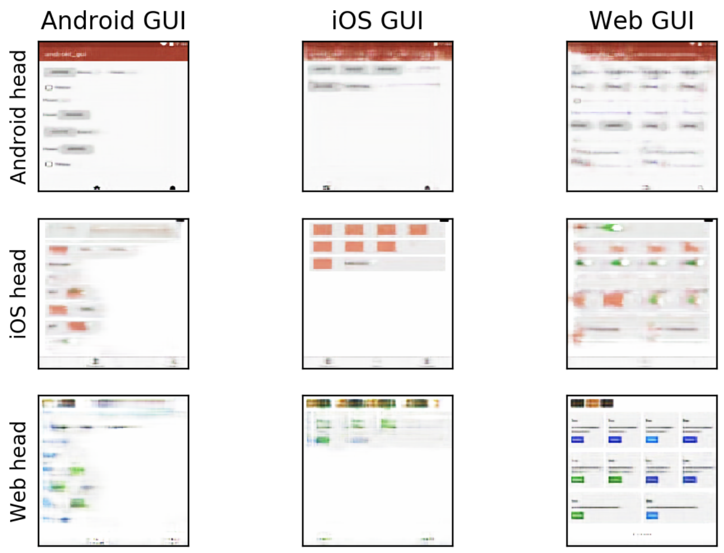


The specific autoencoder design we used shared the same image encoder between all three image decoders. This ended up being a very efficient model, and produced generalizable image decoders.

We use the decoder heads in the Code2Pix model design. To see the specific underlying architecture of the encoder and decoder heads, we point the reader to our Github repo.

Towards Universal Cross-Platform User Interfaces

Our Hydra autoencoder learned embeddings which give credence to the possibility of systems which can compile universal user interfaces. The following image is what 3 images would look like on different platforms, according to our autoencoder. For example, the leftmost column is what an Android GUI would look like on iOS and Web.

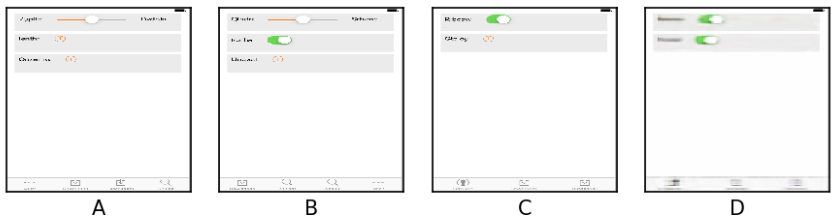


Three different images in three columns being interpreted similarly between each platform. The left column is an Android image, middle is an iOS image, and the right column is a Web image. Each row represents what that image would look like on a different platform.

The left and right columns show a surprising degree of structural similarity. In the left column, the iOS head sees the Android buttons as orange boxes, and the web head sees the same buttons as green buttons. In the right column, we can see that both the Android and iOS head capture the essential segmented structure of the Web GUI, using combinations of buttons and sliders to replicate the Web's div elements.

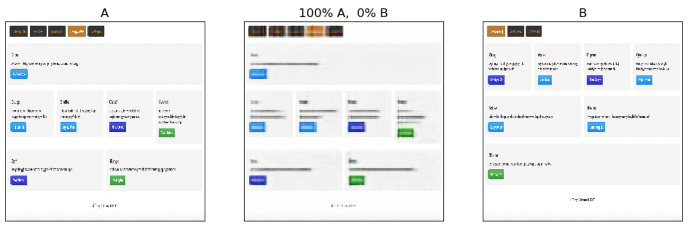
The fact that each image still has structure preserved when interpreted to new platforms gives hope that there might one day be a deep learning compiler capable of seamlessly translating user interface code between Android, iOS, and Web backends. This is a very simple, unrefined model, yet with more time and experience, it's possible it could produce more sophisticated results. These results show that the encoding space learned by the autoencoder has enough structure to be useful in future applications.

To validate the structure of the embedding space, we query the model to interpolate between different images. Drawing off of the famous “King Man + Woman = Queen” paper that visualized how models learn to interpret word data, we did our own “user interface algebra.”



”A is to B as C is to D.” Here we compute B—A + C = D in order to obtain the result above. The main difference between B and A is the toggle in the second row. By adding B—A to C, we can see D now has a button in the second row.

We do note that not every combination of images works as cleanly as the example above. However, this is just the first step at something larger: true user interface algebra and a GUI compiler that works across Android, iOS, and Web interfaces.

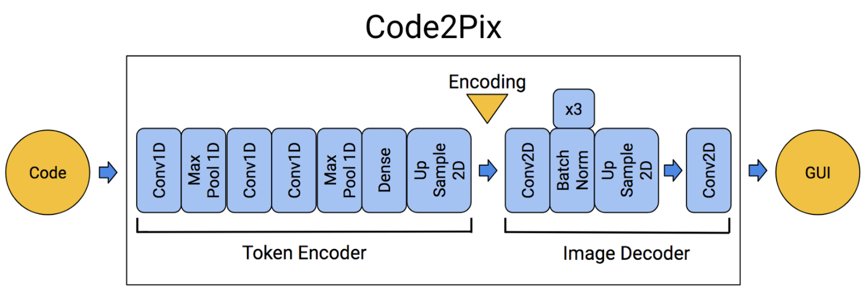


More Eyecandy: Using the Web decoder to translate between two different Web GUIs.

Code2Pix Model Design

After having analyzed the encodings of the autoencoder, we were prepared to move onto the final step. We wanted to frame the model design process as an encoder-decoder dynamic. There had to be a portion of the model that interpreted the code tokens, and it had to communicate with a model that decodes the encodings into the actual picture again.

We started by attempting to import pretrained LSTMs (deep learning layers which are great at learning sequential data) from the Pix2Code model, but that wasn't showing any promise. We then experimented with convolutional layers, and ended up using 1D convolutional layers to encode the code tokens from the domain specific language. We train our model by converting each DSL token of the sequence to a one-hot representation, then concatenate them to create a matrix that we can then feed to the Code2Pix model. We zero pad sequences such that they all have the same length.



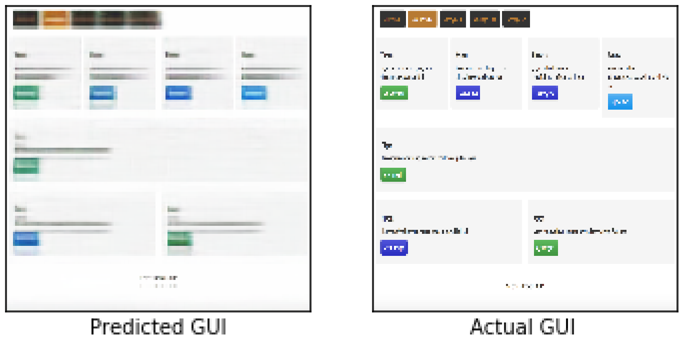
Final architecture for the Code2Pix model.

4—Code2Pix Results

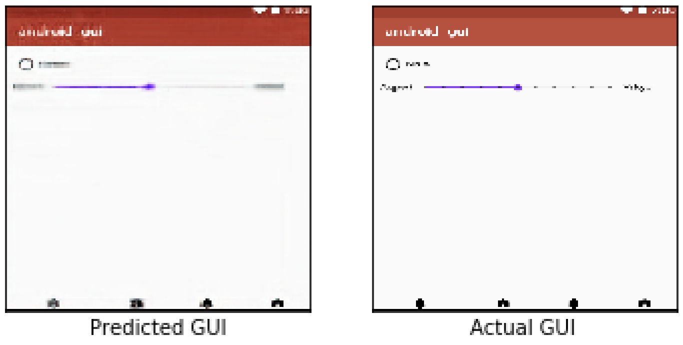
This model is very fast to train, and produces surprisingly clean output images for all three images. It takes around 55 epochs to train Code2Pix for a single dataset, which is about 30 minutes on two Nvidia GTX 1080s. Interestingly the model does equally well on the iOS and Web datasets, but struggles to generate fine-grained details for the Android dataset.

A surprising result from this final architecture was that the pretrained image decoder did not help train the Code2Pix model at all. In fact, it was *just as fast to train* the model without the pretrained weights as it was *with the pretrained weights*. A natural conclusion would be that this is a very stable model architecture for this task and dataset. A follow-up hypothesis would be that the encoded vector space for the pixel-to-pixel problem is actually different from the encoded vector space for the text-to-pixel problem. As a result, using pretrained weights on the GUI-to-GUI problem does not provide valuable information for the model to learn faster on the DSL-to-GUI problem.

The following pictures are selected examples from our validation dataset.



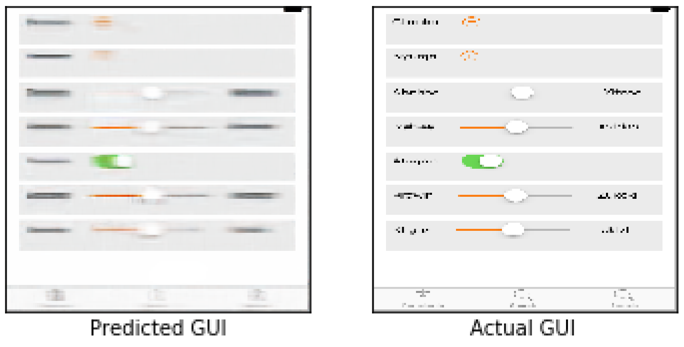
(Web) Corresponding code: header { btn-inactive btn-active btn-inactive btn-inactive btn-inactive } row { quadruple { small-title text btn-green } quadruple { small-title text btn-red } quadruple { small-title text btn-red } quadruple { small-title text btn-orange } } row { single { small-title text btn-green } } row { double { small-title text btn-red } double { small-title text btn-green } }



(Android) Corresponding code: stack { row { radio } row { label slider label } } footer { btn-notifications btn-home btn-notifications btn-home }

We were really happy when we saw this. The textual description given for this model is relatively complex, and the picture is also relatively complex—yet the model nailed it. Given how simple the Code2Pix model is, the fact that it can reach this high resolution is quite surprising.

Code2Pix had the most trouble generating clear resolution images of the Android images. It is possible that the Android images may have more intricate details than the other GUIs, like the dots on the slider.



(iOS) Corresponding code: stack { row { label btn-add } row { label btn-add } row { label slider label } row { label slider label } row { label switch } row { label slider label } row { label slider label } } footer { btn-download btn-search btn-search }

The iOS images are much cleaner than that of Android's. Additionally, the iOS Code2Pix model is capable of handling more complex textual descriptions.

5—Limitations

The expressiveness of our deep learning renderer is constrained by the datasets and the domain specific language. The DSL has to be hand-crafted along with a synthesized dataset to expand the amount and variety of UI elements the deep learning compiler can support. Additionally, the images generated by the deep learning compiler are not as clear as the true images. I think this has to do with the model learning to "hedge its bets" on some ambiguous looking UI elements, like sliders. It's possible this is just an artifact that can be removed with some fine-tuning or more data.

But the thing is, these experiments are just the beginning. Working on these projects feels like learning how to perform chemistry experiments. Right now, we're just combining hydrogen and oxygen to make water, but we have so much further to go. We haven't even gotten to working with rocket fuel yet. If Uizard's demo shows us anything, it's that there is so much more in store for us in the future.

6—Looking Back: Mistakes Made, Lessons Learned

Working on this project, we relearned the importance of fundamentals. In solving Code2Pix, we did bottom-up development, building autoencoders that we ended up not necessarily needing in the final model. We could have started by constructing a baseline model for our Code2Pix. This could have sped up our model development process.

Additionally, the biggest mistake I personally made was a technical blunder. In developing the final Code2Pix model, I failed to verify that the data was being loaded in properly. It turns out the (x, y) pairs were being mismatched upon loading. This cost us somewhere from 2–3 weeks of precious model design and application refinement time. What'd I learn? Unit test the data pipeline process, or use another validation process. For example, I could have just imported Pix2Code's data pipeline process. Instead, our research code had 3–4 different ways of loading in the data, and unfortunately only 2–3 of them worked. Not making that mistake again.

In summary, the more important lessons I learned were soft ones that are harder to articulate and are now embedded in my work ethic. Tweaks to my communication skills, reduction in micro-management, the sometimes uneven development of trust, and stuff like controlling my verbosity setting in team meetings. These were the fun lessons!

Acknowledgements

Big shoutout to my team at SUSA for believing in my leadership and putting hours of great work into this. To Samyak Parajuli, Luke Dai, Ajay Raj, Aismit Das, Dennis Yang, and Japjot Singh—thank you, this was a load of fun to work on with you guys.

We would not have done this project without Tony Beltramelli's great guidance and mentorship. Thank you Tony for mentoring us!

· · ·

Thank you for reading! If you have more questions about our research, feel free to email me at `noah.gundotra@berkeley.edu` . If you're interested in working at Uizard, see their career page!