Kaunas University of Technology

Faculty of Informatics

Applying DNN for HTML Generation from Webpage Screenshot

Master's Final Degree Project

Povilas Simanaitis

Project author

Prof. Rytis Maskeliūnas

Supervisor

Kaunas, 2021

Kaunas University of Technology

Faculty of Informatics

Applying DNN for HTML Generation from Webpage Screenshot

Master's Final Degree Project /

Informatics (62111BX007)

Povilas Simanaitis

Project author

Prof. Rytis Maskeliūnas

Supervisor

Dr. Šarūnas Packevčius

Reviewer

Kaunas, 2021

Kaunas University of Technology

Faculty of informatics

Povilas Simanaitis

Applying DNN for HTML Generation from Webpage Screenshot

Declaration of Academic Integrity

I confirm that the final project of mine, Povilas Simanaitis, on the topic "Applying DNN for HTML generation from webpage screenshot "is written completely by myself; all the provided data and research results are correct and have been obtained honestly. None of the parts of this thesis has been plagiarised from any printed, Internet-based or otherwise recorded sources. All direct and indirect quotations from external resources are indicated in the list of references. No monetary funds (unless required by Law) have been paid to anyone for any contribution to this project.

I fully and completely understand that any discovery of any manifestations/case/facts of dishonesty inevitably results in me incurring a penalty according to the procedure(s) effective at Kaunas University of Technology.

_____	_____
(name and surname filled in by hand)	(signature)

Povilas Simanaitis. Applying DNN for HTML generation from webpage screenshot. Master's Final Degree Project supervisor Prof. Rytis Maskeliūnas; Faculty of Informatics, Kaunas University of Technology.

Study field and area (study field group): Informatics, Physical sciences

Keywords: pix2code, multimodal-space, Html, code generation.

Kaunas, 2021. 82 pages.

Summary

In this paper, we apply deep neural networks for generating HTML code from a webpage screenshot. First, synthesise our dataset in which web page screenshots are like pix2code, but differently from pix2code, we use a plain Html instead of a domain-specific language, which increases task complexity. While still following encoder-decoder network architecture, we replace decoders LSTM architecture with transformer-based architecture, and for some of the experiments, we also replace CNN based encoder with Transformer. Thus our applied model either follow image captioning with stacked attention architecture or full transformer architecture for image captioning. Using these newer architectures allows us to achieve better accuracy than pix2code authors.

Povilas Simanaitis Hiperteksto žymėjimo kalbos dokumento rengimas, pagal vaizdą naudojant dirbtinį intelektą. Magistro baigiamasis projektas vadovas Prof. Rytis Maskeliūnas; Kauno technologijos universitetas, informatikos fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): Informatika, Fiziniai mokslai.

Reikšminiai žodžiai: pix2code, Html, kodo generavimas.

Kaunas, 2021, 82 p.

Santrauka

Šiame darbe yra apžvelgiama giliųjų neurotinių tinkle taikymas kuriant HTML kodą pagal saityno svetainės nuotrauka. Mes sukūrėme sintetinį duomenų sąrašą, kurio paveiksliukai yra panašūs į pix2code duomenų sąrašo, tačiau skirtingai nuo pix2code naudojamos domeno kalbos mūsų duomenų sąrašas naudoja HTML. Šis sekos pakeitimas padidina sekos ilgį ir sekoje panaudotų žodžių žodyną.Mes naudojama užkuoduotojo-atkoduotojo tinkle struktūrą, tačiau pakeičiam atkoduotoją “transformer” pagrindo tinklu, nes anksčiau naudotas ilgalaikės trumpalaikės atminties tinklas negeba teisingai nuspėti ilgų sekų. Mes taip pat išbandome keturis skirtingus tinklus, viename iš jų sąsukų tinklai užkoduotojoje pakeičiami vaizdo transformerio tinklu. Panaudojant šiuos tinklus pasiekiama taiklesnis spėjimas atkuriant sekas tiek ir su mūsų,tiek ir su pix2code duomenų sekomis.

Table of contents

List of tables

List of abbreviations and terms

Abbreviations:

LSTM. – Long Short-Term Memory.

GUI. – Graphical user interface.

DSL. – Domain-specific language.

CNN. – Convolutional neural network.

DNN. – Deep neural network.

RNN. – Recurrent neural network.

HTML. – Hypertext Markup Language.

API. – Application programming interface.

CNTK. – Microsoft cognitive toolkit.

NAS. – Neural architecture search.

GUI – Graphic user interface

PWA – Progressive web app

WBA – A web-based application

GRU – Gated recurrent unit

Introduction

Creating software to this day is mainly done by humans; however, recent advancements in the machine learning field hint that some parts could be replaced by applying machine learning solutions.

Relevance of the project.

Web-based applications are becoming more widely used and acknowledged. Since progressive web apps (PWAs) became distributable via App Store and Google Play store, they started to compete with native mobile applications. Even before introducing PWA, web-based applications (WBA) were making their way as desktop applications, replacing cross-platform applications based on virtual machine runtimes such as JAVA or Visual Basic. There is even an operating system Chrome-OS, which primarily utilises WBAs. Recent changes, which enabled applications to work offline, access device hardware, other features accessible only to native apps, and WBA's ability to work on multiple operating systems, made WBA a common choice among software developers.

All WBAs have two things in common; they use JavaScript and Hypertext Markup Language (HTML). Until this day, most HTML is created by humans. There is little research done in trying to automate this task. Our project aims to apply a deep neural network to automate HTML creation.

Our produced software and investigation could be used for further research. Data-synthesizer, synthesising larger and more complex datasets. Although every internet page can be treated as a data instance, the HTML standard receives constant updates, making not every website compatible with the current standard. Thus, a dataset synthesiser provides a way to configure and generate clean datasets which follow the current HTML standard datasets.

Purpose and objectives

Our main goal is to investigate Deep Neural Network (DNN) appliances for generating HTML by creating a model that would convert images to HTML. Achieving this goal can be split into the following objectives:

1. Analyse image processing, language modelling, and captioning image models and find the best theoretical architecture for generating HTML from webpage screenshots.
2. Create software that would generate a pix2html dataset (similar to pix2code in image part, but uses HTML instead of DSL in sequence part)
3. Create various DNN capable of generating HTML from a webpage screenshot and investigate their results.
4. Compare our solution with already present solutions.

Document Structure

The rest of this paper consists of five main sections that describe our investigation, projecting, implementation, and experimentation done while creating our model and all related software. In the first section, we analyse related problems and their solutions by overviewing related research. We do an overview of various Convolutional Neural Networks (CNN) improvements over the past decade, typical image captioning techniques, and related images to code conversion papers and the evolution of sequence prediction-related networks. In the second section, we describe the requirements and design of our solution (model and additional dataset generating software) needed to achieve and their various UML diagrams. The third section focuses on implementing our solution. And its performance,

measured by tweaking different hyperparameters and executing multiple experiments. In section four, we describe various experiments we have performed while doing research. In the fifth, final section, we conclude, Outlining the most relevant results and discoveries of our research.

1. Analysis

This part is made from five subparts. In the first subpart, we overview the history of image processing networks. In the second one, we overview language processing models. In the third, we overview image captioning models. Furthermore, in the fourth subpart, we overview work related to code generation from images.

1.1. Various image processing architectures

CNN has initially introduced 30 years ago as a solution for handwritten zip code recognition (1) and, ever since, have been many advancements by applying various techniques. CNN has at least one hidden convolutional layer, which uses filters, which are convoluted either through the whole image or the results of previous convolution layers. Thus the name convolutional neural network.

1.1.1. Alexnet

Introduced in 2012, he popularised two novelties in CNN, Rectified Linear Unit (ReLU) activation function, which allowed their network to train faster than a network that uses sigmoid activation. Furthermore, it also introduced overlapping pooling. They observed that the model which uses overlapping pooling is less likely to overfit. Alexnet has eight layers, with the first five being convolutional and the rest three fully connected. (2)

1.1.2. VGGnet

VGGnet architecture was introduced as an improvement for conventional convolutional networks. It focused on the importance of depth in visual representation by having more convolutional layers. Traditional VGG networks are either 16 or 19 layers, which in 2015 were very deep. Exact network configurations *Fig. 1 VGGnet layers* (3)

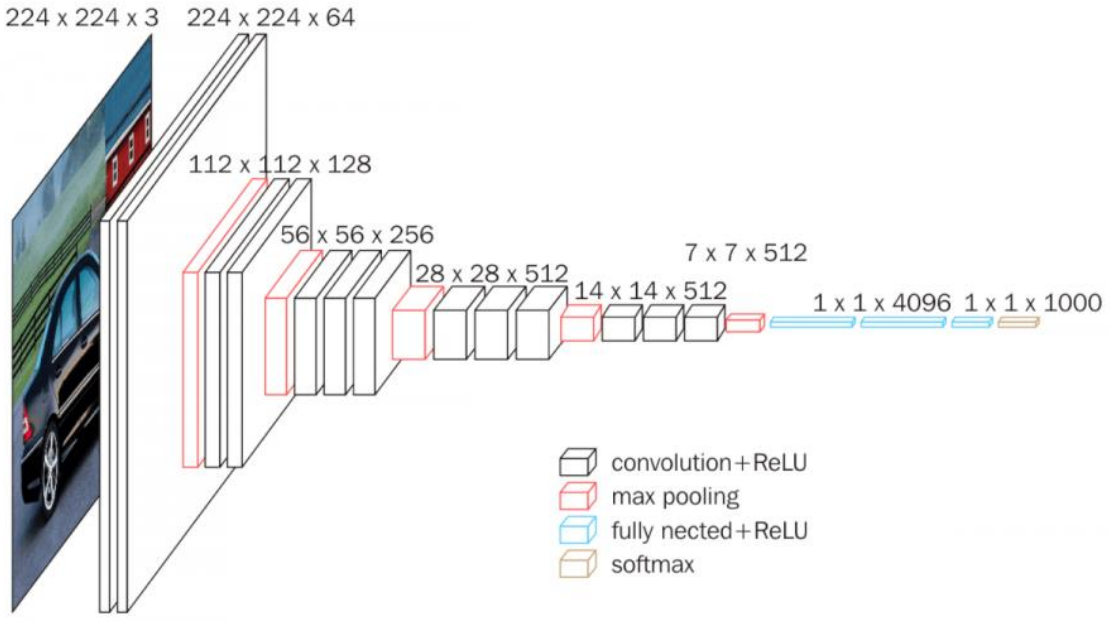

Fig. 1 VGGnet layers visualised

1.1.3. Efficient Net

CNN architecture is usually improved from previous state-of-the-art architecture by either making network deeper, increasing the network's layer size, or accepting higher resolution images as shown in *Fig. 2 Various CNN scaling* techniques. This paper investigates various combinations of network improvements and finds compound coefficients. That enables us to determine the most effective way to scale a network. They scale various networks while achieving better results and at the same having smaller networks. Their compound coefficient could is expressed as $a * b^2 * c^2 \approx 2$ where a $a \geq 1$ b ≥ 1 c ≥ 1 and a stand for network width, b stands for network depth, and c stands for input image resolution. (4)

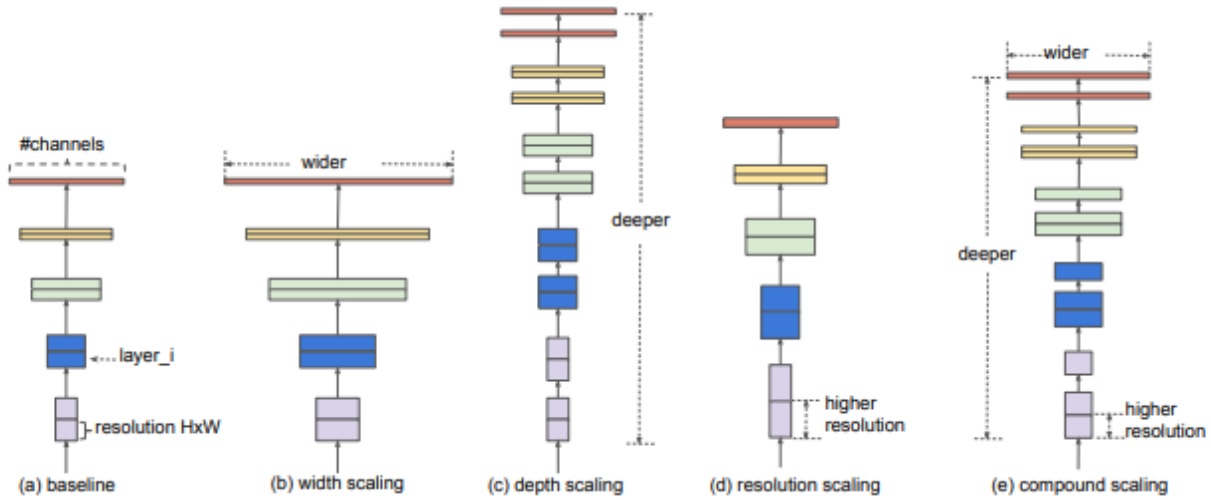

Fig. 2 Various CNN scaling techniques

1.1.4. Resnet

Resnet architecture aims to solve the issue that appears when trying to train networks, which have many layers, thus having vanishing/exploding gradients. It achieves lower complexity than VGG16 even with the network, which has up to 8 times more layers. This network overcomes vanishing exploding gradients by introducing residual block, which enables to connect to two not sequential layers directly via identity shortcut connection (marked as x identity in *Fig. 3 Shortcut connection visualisation*). (3)

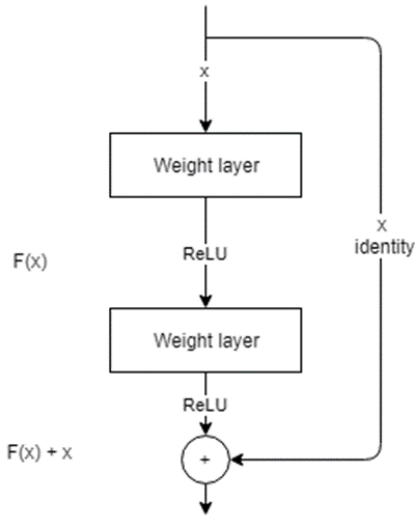

Fig. 3 Shortcut connection visualisation

1.1.5. Inception

Inception architecture was inspired by the network in network architecture, which adds a 1x1 convolutional kernel to reduce dimensions. Inception takes this several steps forward by having multiple convolutions in a single layer, as shown in *Fig. 4 Inception module with dimensions reduction* (5)

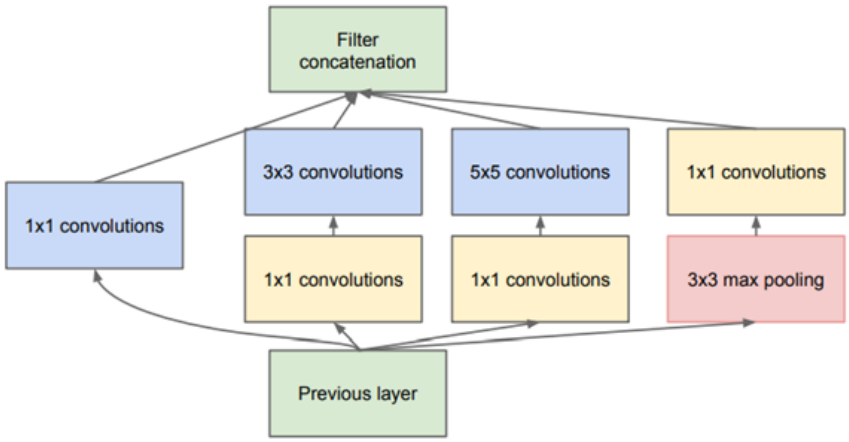

Fig. 4 Inception module with dimensions reduction

Also, there was an improvement for this architecture called InceptionV2. It was achieved by firstly replaced 5x5 convolution with two 3x3 convolutions, improving speed by 2.78 times. Then 3x3 convolutions were replaced with a layer of 1x3 and 3x1 convolution, further enhancing the performance of the inception module by 33%. InceptionV3 introduces an additional 7x7 kernel which is factorized (split into 1x7 and 7x1 convolutions) (6)

1.1.6. DenseNet

DenseNet introduces a new connection between layers pattern, where each layer has a direct link from all subsequent layers, as shown in *Fig. 5 Dense connections visualised*.**Fig. 5** Dense connections However, differently from Resnet, it does not combine features via summation. They combine them via concatenation. Authors reduced network complexity by using transition layers between 2x2 average pooling with a stride of two and a layer of 1x1 convolution. Thus, enabling to perform deep supervised learning, which resulted in better whole network accuracy. (7)

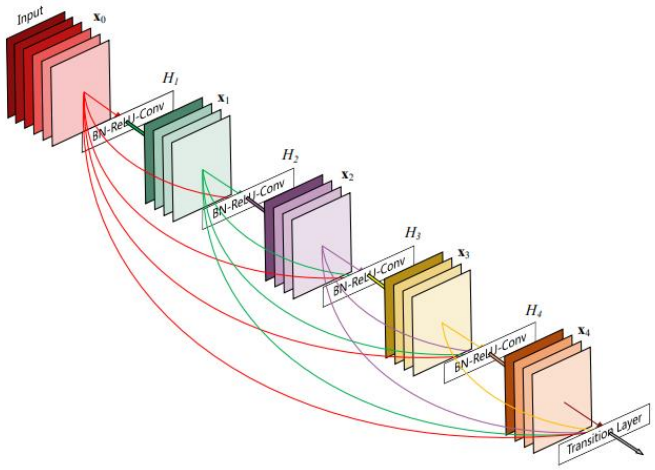

Fig. 5 Dense connections visualised

1.1.7. Xception

Xception (meaning extreme Inception) proposed a CNN based entirely on depth-wise separable convolution layers. It replaces inception modules with extreme inception modules, as shown in *Fig. 6 Xception module*. It shows slight performance gain in terms of accuracy over inceptionV3 (8)

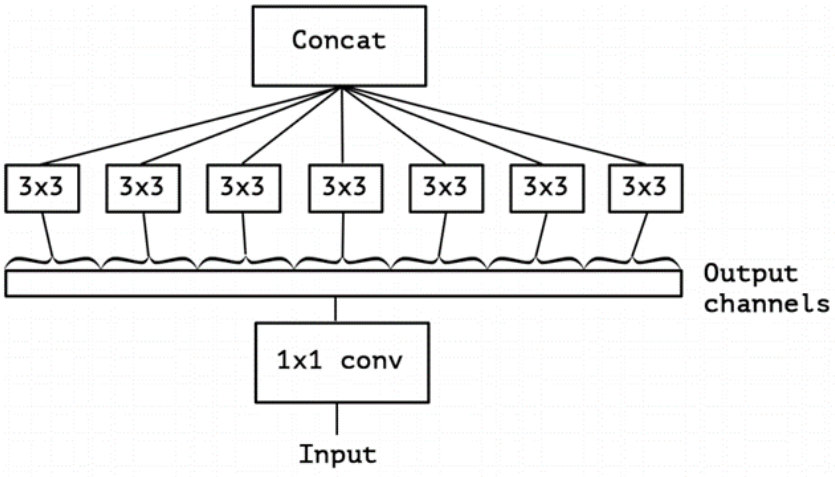

Fig. 6 Xception module

1.1.8. DilatedNet

Dilated Residual Networks focused on applying dilation instead of subsampling. Their research shows that replacing subsampling with dilation models could result in a more accurate model without adding additional layers or making the whole model more complex. By applying dilation, they try to preserve spatial information. However, dilation may cause grinding artefacts, and in their research, they find out that max-pooling can exacerbate these artefacts. Therefore, they replaced max pooling with convolutional filters, then they added more layers with lower dilation, and finally, they removed residual connection in the last layers. Thus, creating a network with having the same number of parameters but higher accuracy. This kind of architecture network is commonly used for landscape recognition. (9)

1.1.9. Nasnet

In Nasnet RNN based controller selects building blocks to create end-to-end architecture. Even though its final structure is already predefined, the algorithm figures out exact blocks to be fitted inside the chosen structure; this generally results in similar to ResNets or DenseNets architectures. It combines different combination and configuration of commonly used image recognition building blocks. Since discovering network architecture is a computationally demanding process, authors trained networks on small datasets and then transferred that architecture to big datasets. Their research results prove that transferring architecture does not result in a significant loss of accuracy. (10) (11)

1.1.10. Resnet Inception

InceptionResnet is similar to inceptionV3. However, it uses different residential connections, as shown in *Fig. 7 Inception resnet p n module*. It has a more negligible computational cost than inceptionV3 while being slightly more accurate than inceptionV3. (12)

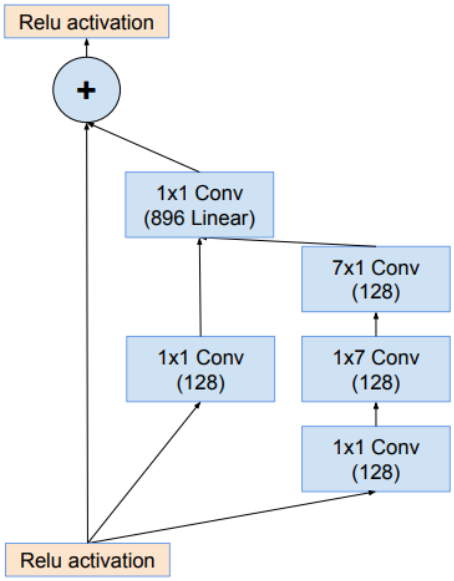

Fig. 7 Inception resnet p n module

1.1.11. Vision transformer

It replaces convolutional layers with attention-based architecture. Though results vary between datasets, Vision transformer is still outperformed by Resnet on a smaller dataset but given a larger dataset (14M-300M images), researchers at Google found it to top its CNN-based rivals. Vison transformer works by first cutting an image into patches. Then it patches are linearly embedded and passed to a standard Transformer encoder while adding position embeddings as shown in *Fig. 8 Image cutting into patches and feeding into VIT* .(13)

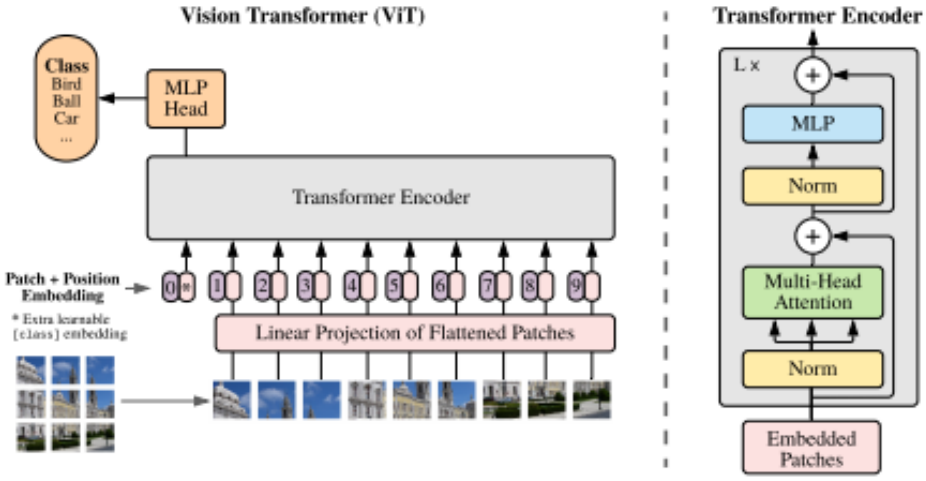

Fig. 8 Image cutting into patches and feeding into VIT encoder

1.1.12. Image processing networks comparison

After various image processing, network analysis, we notice that accuracy significantly increases by increasing network width, resolution, depth, introducing new types of cells, or new connections between network nodes or layers. Also, according to *Table 1 Image processing networks comparison*, we can generally say that having more parameters results in higher accuracy, and the highest network, which has 224x224x3 dimensions, is Resnet.

Table 1 Image processing networks comparison

Architecture name	Input dimensions	Number parameters	ImageNet accuracy
Alexnet	256x256x3	60M	63.3 %
VGGnet-16	224x224x3	138M	74.4 %
EfficientNet-B0	224x224x3	5.3M	76.3 %
Resnet	222x224x3	25M	77.15 %
Inception-V2	299x299x3	11.2M	77.8 %
Xception	299x299x3	22.8M	79 %
ResnetInception	299x299x3	55.8M	80.1 %
Vision Transformer	256x256x3	307M	87.76 %

1.2.Various language predicting models.

Sequence prediction plays an essential part in our model. Generating long sequences was always a challenge for deep neural networks. In this section, we will overview the evolution of sequence prediction models.

1.2.1. RNN

RNN was introduced back in 1985 by David E. Rumelhart, and James L. McClelland in their article Learning Internal Representations by Error Propagation. They introduced an autoregressive network, which means that it uses self-previous outputs as a part of the input for the next prediction. Thus, the network can take a sequence as an input and produce a sequence as an output. Their suggested network is displayed in *Fig. 9 RNN network*. where t means another time step and U its output. (14)

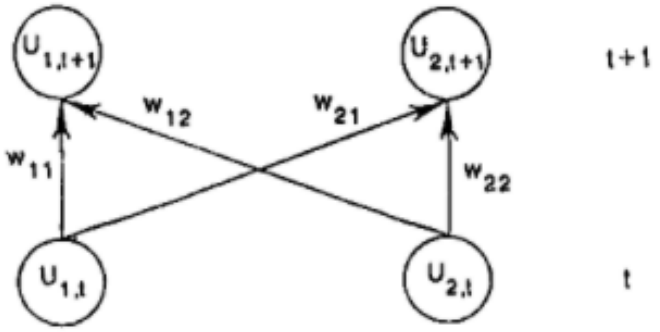

Fig. 9 RNN network

1.2.2. LSTM

Even though RNN could use sequential data for prediction, the order of data entries inside sequence played a crucial part, meaning that neighbouring data entries impacted prediction a lot more than data entries that are further away. The difference in impact on the next prediction is known as the exploding-vanishing gradient problem. To solve it, Sepp Hochreiter, Jürgen Schmidhuber introduced an LSTM cell with an additional forget gate as shown in *Fig. 10 LSTM cell visualised.* Forget gate discovers which details can be discarded from the cell. That is decided via sigmoid function by looking at the previous state (ht-1) and the input(Xt) and outputs for each cell state Ct−1. (15)

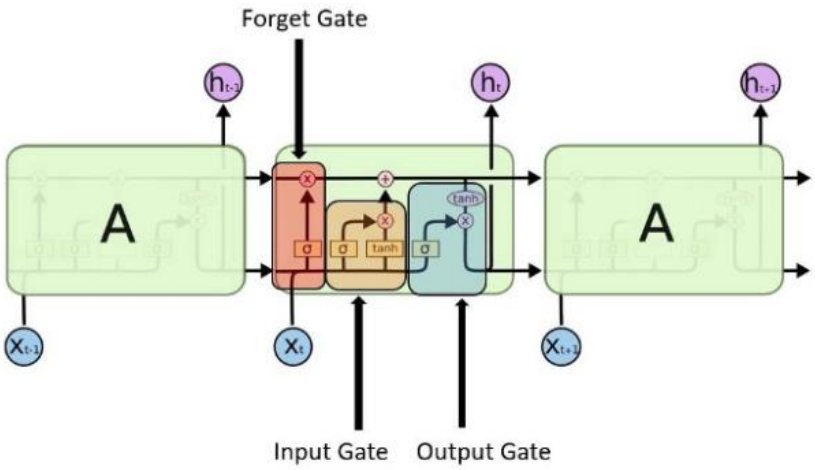

Fig. 10 LSTM cell visualised.

1.2.3. GRU

Gated recurrent unit (GRU) aims to solve the same vanishing gradient problem, though it was introduced 17 years later than LSTM. GRU has two additional reset and update gates while retaining input and output gates. That enables a cell to keep information from a long time ago without losing it through time or removing information irrelevant to the prediction. Their proposed activation is displayed in *Fig. 11 GRU network cell* where the update gate, z, selects whether the hidden state should update with a new hidden state h˜. The reset gate r decides whether the previous hidden state is ignored (16)

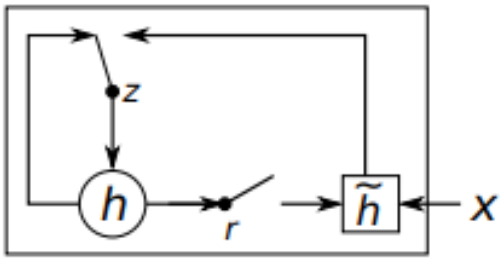

Fig. 11 GRU network cell

1.2.4. Sequence to Sequence

This architecture uses encoder-decoder architecture, where both encoder and decoder are recurrent neural networks, the encoder extract information to a context vector from which is then used by the decoder to predict. It aims to solve an issue with different lengths of input and output sequence. However, since the context vector is a fixed length, this network cannot remember long sequences. (17)

1.2.5. Attention mechanism.

It aims to help the seq2seq model to solve the long source sentence problem. Instead of building a single context vector out of the encoder's last hidden states, it creates shortcuts (called global alignment scores shown in *Fig. 12 Attention mechanism)* between the context vector and the entire input. The weight of each of these shortcuts is trained together with the network. Decoder attends over the sum of hidden states weighted by alignment scores. (18)

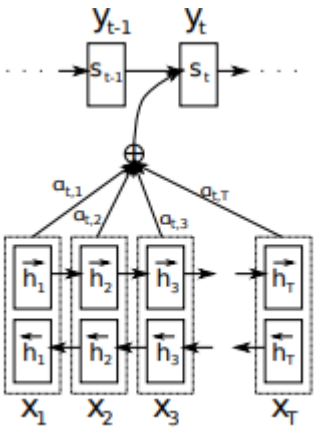

Fig. 12 Attention mechanism

1.2.6. Transformer

Transformer architecture proposes to ditch recurrent neural networks in favour of attention. It eliminates various RNN forms and uses positional encoding (sometimes referred to as positional embedding) and multiheaded attention. Multi-head attention aims to establish a connection between long-distance dependencies, while positional encoding helps the network to understand the importance of sequence order. Multi-head attention used in the first layer of the encoder is called self-attention because all parts of it come from the same place - the previous layer. Multi-head attention in the first layer of the decoder is similar to encoders' self-attention. However, to preserve auto-regressive property, they implement scaled dot product attention that masks SoftMax inputs. This masking is needed because of the teacher-forcing used to train this network, and if the masking is not present, the network results in illegal connections. The second Multi-head attention inside decoder queries comes from the first subpart of the decoder. Still, keys and values come from the encoder, thus creating combined attention on both encoder-decoder parts. The whole transformer architecture is displayed in *Fig. 13 Transformer architecture* (19)

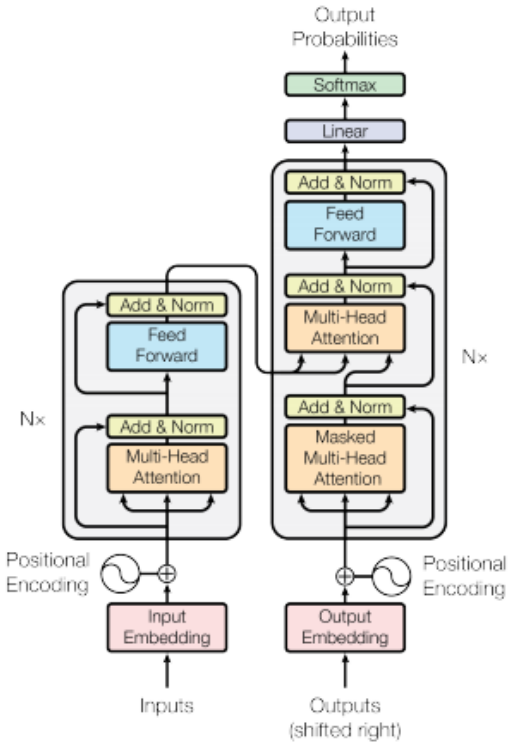

Fig. 13 Transformer architecture

1.2.7. Denoising autoencoders

It uses a non-autoregressive model, which uses latent variables to predict sequence tokens independently from other predictions. That model enables the parallelisation of the training process. That is impossible when training LSTM with an attention network because forward and backwards pass through the sequence during training. Furthermore, when using a transformer-based, it is not possible to parallelise decoding at the test time. Denoising autoencoder uses simple supervised optimisation to calculate latent variables and minimising cross-entropy between ground truth and predictions. They also use distillation, a good autoregressive model prediction, as a target to their non-autoregressive model. It allowed them to improve their results by 10-15%. Their architecture is based on a transformer. However, they do have two decoders instead of one. Using adaptive training second decoder keeps on running until the output stops changing. The whole network architecture is shown in *Fig. 14 Denoising autoencoder*. (20)

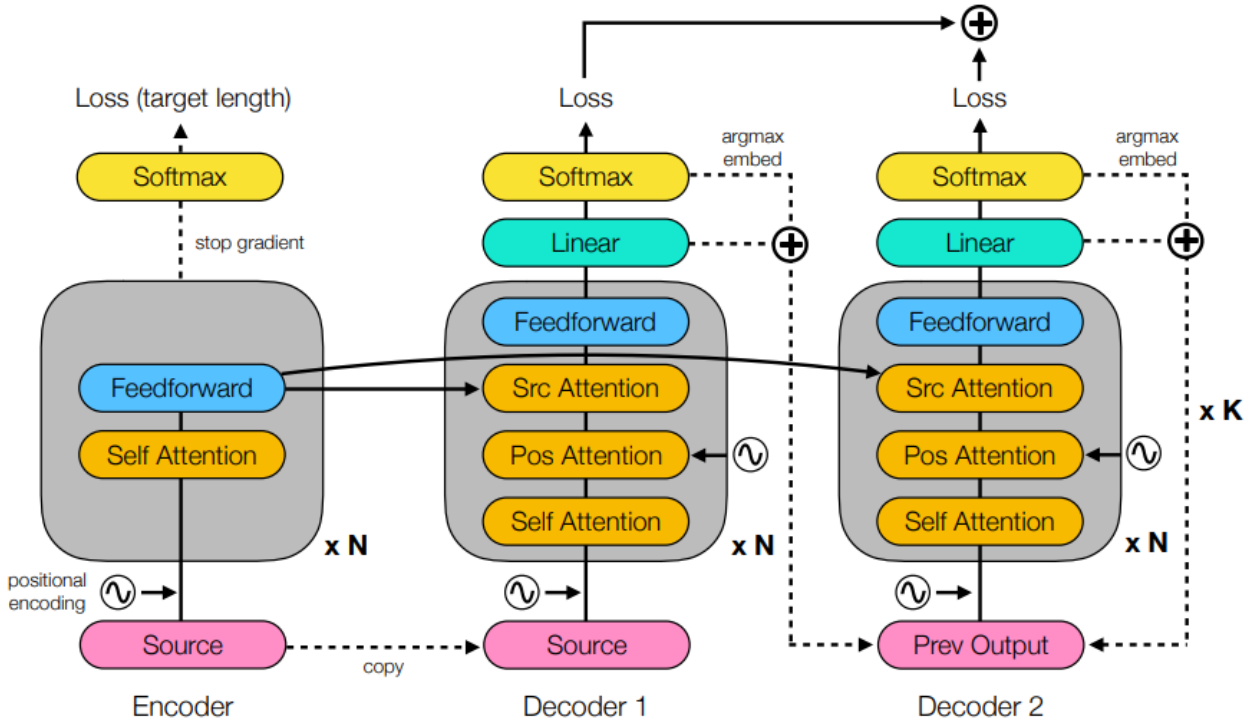

Fig. 14 Denoising autoencoder

1.2.8. Reformer

Even though the transformer can achieve the state-of-the-art result, its computation cost rapidly grows when increasing input size. It makes it hard to train models with long sequence inputs on the public accessible computing device. It introduces two significant enhancements to the transformer model. First, they changed dot product attention with locality-sensitive hashing, thus only calculating nearest neighbours' attention rather than with the whole sequence. Second, they added reversible layers. They removed a necessity to store activations for backward propagation. Thought increased calculation time, but it reduced needed memory even further. The reversible layer is displayed in *Fig. 15 Reversible and standard connection visual comparison* (21)

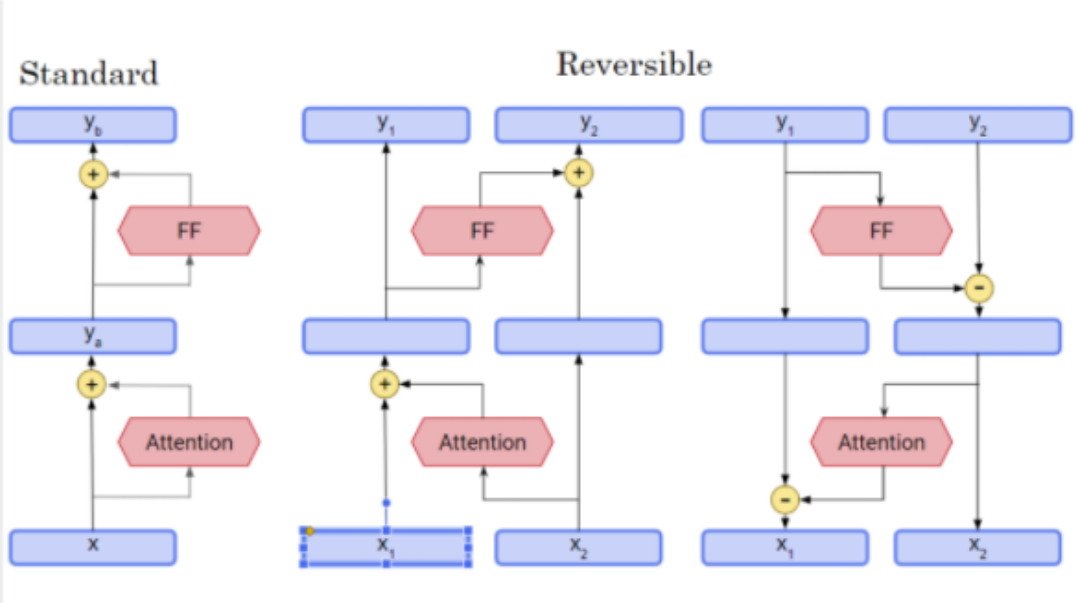

Fig. 15 Reversible and standard connection visual comparison

1.2.9. Sequence modelling models comparison

Various sequence processing network analysis indicates that using attention allows for better results and that the original LSTM shortcoming inability to deal with long sequences is solved. According to *Table 2 Language processing networks comparison on WMT2014-English german dataset*, Reformer yields the best results when translating from English to German; however, the accuracy increases just by 1.8 BLEU. At the same time, calculations get significant increase.

Table 2 Language processing networks comparison on WMT2014-English german dataset

Architecture name	**BLEU score**
Denoising autoencoder	21.54
Transformer	27.3
Reformer	29.1

1.3.Image captioning architectures predicting models.

The task of generating code from the image is very similar to image captioning. Image captioning is a combination of computer vision and language modelling problems. Captioning models follow encoder-decoder architecture. In this section, we will overview the evolution of image captioning models

1.3.1. Encoder decoder architecture.

Today, image captioning models implement a decoder-encoder framework, which takes at least two Long Short-Term Memory (LSTM) layers. Where one's final layer becomes the other input layer. In captioning, encoder output is a combination of result from deep CNN which attended over image and LSTM, which attended over the caption, that results in multimodal space vector (22), which is used as input for Decoder an LSTM as shown in *Fig. 16 Image captioning architecture* .(23)

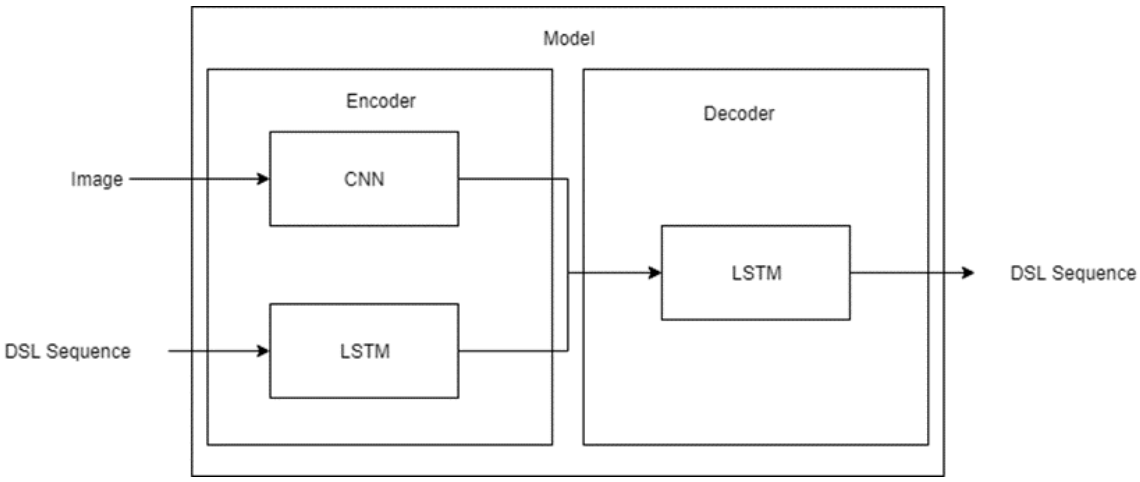

Fig. 16 Image captioning architecture

1.3.2. Show Attend and tell

This paper introduces soft and hard attention-based image captioning generators. Generators follow encoder-decoder architecture, while additionally, they can visualise how a network attends over an image while generating a caption. This work's novelty is that instead of the previously used last fully connected layer of CNN for feature extraction, it uses a previous layer. That allows the decoder to select parts that it wants to "focus" on. Model is visualised in *Fig. 17 Show attend and tell model explanation.*(24)

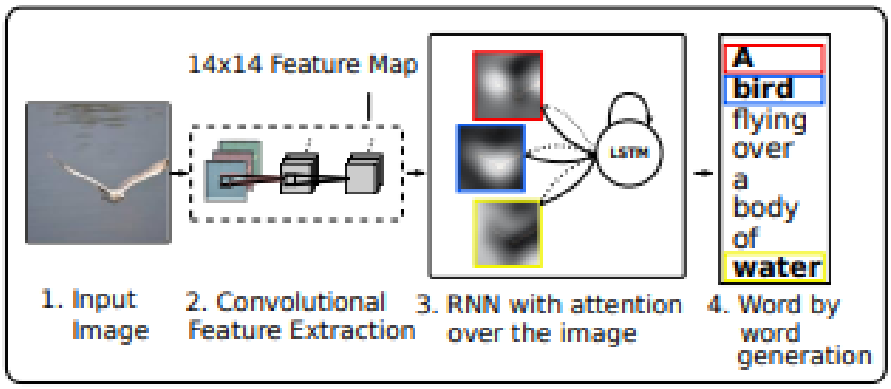

Fig. 17 Show attend and tell model explanation

1.3.3. Captioning Transformer with Stacked Attention Module

Introduce architecture change in which LSTM is replaced with the transformer in encoder-decoder architecture. Their decoder has six layers. They also introduce a new multi-level supervision learning method where each decoder layer output is used to calculate a loss, as shown in *Fig. 18 Captioning transformer loss calculation.* (25)

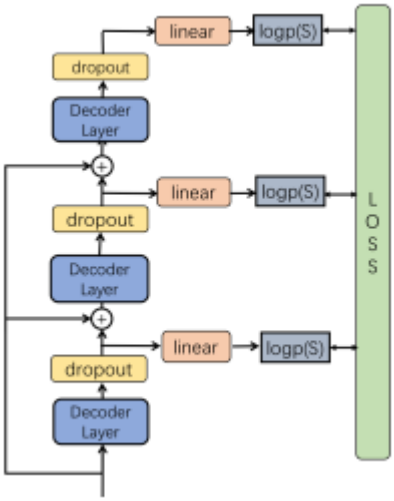

Fig. 18 Captioning transformer loss calculation

1.3.4. Image Captioning: Transforming Objects into Words

This model follows encoder-decoder architecture, where the decoder consists of a standard transformer decoder block. The encoder, rather than image feature vectors as in Captioning transformer with stacked attention, uses features obtained from specific image regions. These regions are obtained by using an image detector, which improves image captioning results. Additionally, they use spatial information obtained from image detectors, such as region size and relative location. They incorporate this relative geometry information into the encoder layer attention part, naming it geometry attention as shown in *Fig. 19 Image captioning with geometry attention* (26)

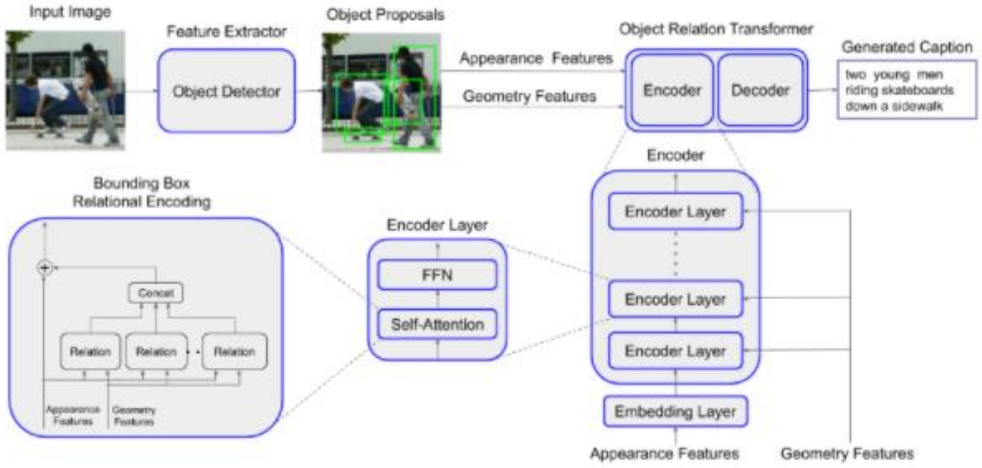

Fig. 19 Image captioning with geometry attention

1.3.5. Meshed memory transformer for image captioning

This network structure replaces LSTM with an attention mechanism too. It encodes image regions in a multi-level fashion, which considers both low-level and high-level relations. Meshed memory transformer introduces a novelty by connecting every encoder layer to decoder layers. These relations contributions are weight at every stage, thus creating mesh connectivity. It outperforms any other state-of-the-art captioning model which uses LSTM. The exact model architecture is shown in *Fig. 20 Meshed memory transformer*. (27)

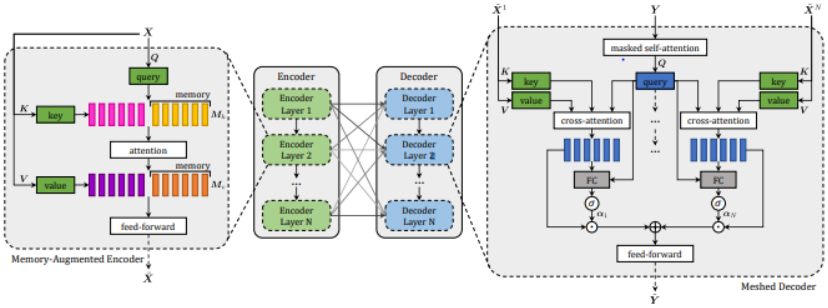

Fig. 20 Meshed memory transformer

1.3.6. Full Transformer Network for Image Captioning

This network replaces CNN used in the encoder with the visual transformer already described in section 1.1.10. Since they use a transformer architecture for the encoder and decoder, it is called full transformer architecture or CPTR. In their work, it demonstrates that it manages to surpass CNN + transformer networks. Networks visualisation is shown in *Fig. 21 Full transformer for image captioning network visualised* (28)

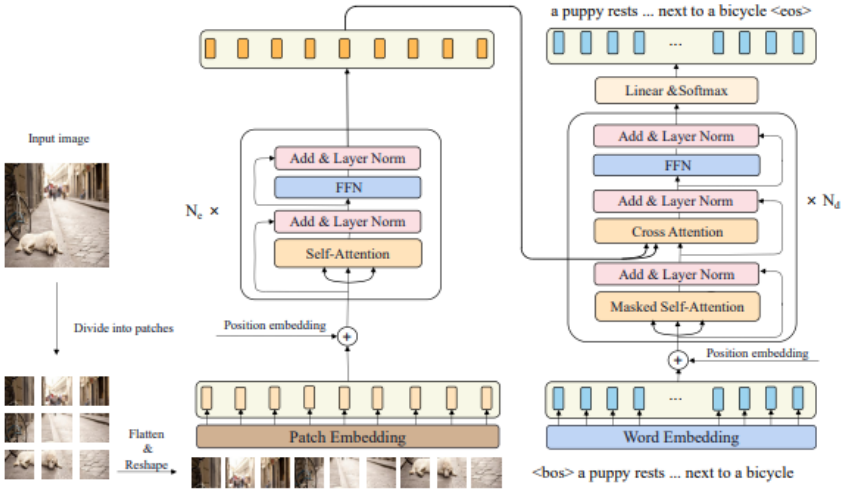

Fig. 21 Full transformer for image captioning network visualised

1.3.7. Image captioning models comparison

All reviewed image captioning models uses encoder-decoder architecture. That enable encoding image features into multi-modal space and decode a sequence from them. The highest accuracy (81.7) on MS COCO is achieved by a full transformer network for image captioning, as shown in *Table 3 Image captioning networks results*

Table 3 Image captioning networks results

Architecture name	**MS COCO BLEU1 score**
Show attend and tell	71.8
Image captioning with stacked transformers	73
Meshed memory transformer	81.6
Full transformer network for image captioning	81.7

1.4. Html generation from image models

There are a couple of papers published on this that are trying to solve this problem. In this section, we will overview both of them.

1.4.1. Pix2Code

Pix2Code model follows encoder-decoder structure, and for image processing, it uses VGG16 architecture. Their language model is made from two LSTM layers, 128 cells each, and their decoder is made from two layers of LSMT, which has 528 cells in each layer. They use 1500 synthesised data entries for training and 250 for validation. Their image dimensions are 224x224. In our upcoming research, we will be following these image dimensions and dataset size. A high-level overview is shown in *Fig. 22 Pix2Code model visualised*. Pix2Code uses DSL to reduce search space. They achieved 88.99% accuracy for recognizing DSL from an image.

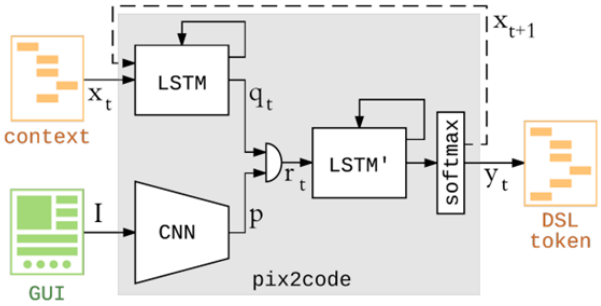

Fig. 22 Pix2Code model visualised

1.4.2. Automatic Graphics Program Generation using Attention-Based Hierarchical Decoder

It uses intermediate filter responses similar to pix2Code and decoder encode framework; however, its novelty is in their proposed Hierarchical visual decoder model. It consists of two models. One LSTM model with a single layer hidden size of 512 cells is used to determine many blocks GUI will consist of and generate guiding vectors for each block. Which then is passed to another model two-layer LSTM model, with a hidden state of 512 cells that creates code tokens. Additionally, they resize images to 256x256 pixels dimension rather than the initial 224x224, and they achieve increased accuracy. Their network is visualised in *Fig. 23 Model with Attention-Based hierarchical decoder* (29)

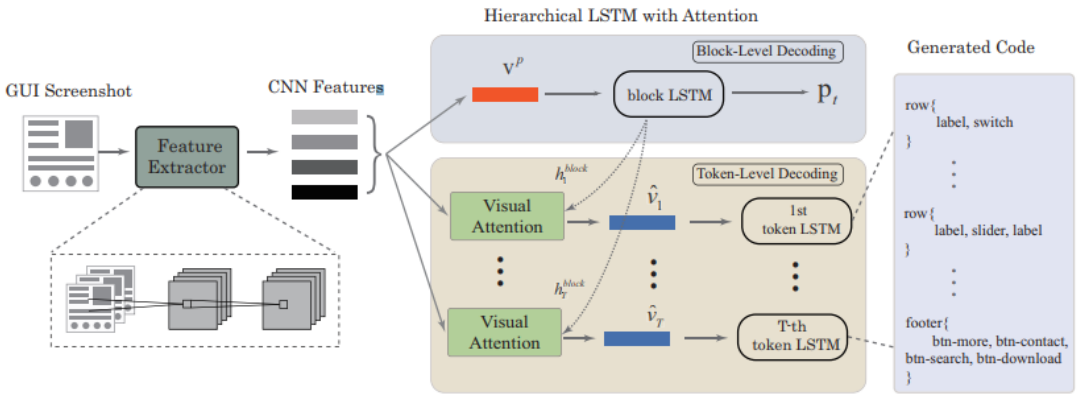

Fig. 23 Model with Attention-Based hierarchical decoder

1.5. Analysis conclusion

After analysis, we will implement encoder-decoder architecture with a transformer used in the decoder because the transformer is shown to deal with long sequences. As for the encoder will either be using a visual transformer or cnn. We will reuse the pix2code dataset and use it as a baseline for our model. Also, we will create a new dataset with 1500 training/validation and 250 testing entries. Then we will compare how our model deals with more extended sequence and see if our model accuracy degrades after increasing models vocabulary and sequence length.

2. Specification of solution

To compare our solution with Pix2Code, we need to generate similar images to Pix2code while replacing DSL with HTML. Thus we need to create a dataset generator. To investigate various model variants, we need to create software that would generate models. Finally, we need to be to train and evaluate multiple models. Therefore, our solution can be split into two main parts: dataset generator and training environment. The high level of our solution use cases is shown in *Fig. 24 Use case diagram of our solution*

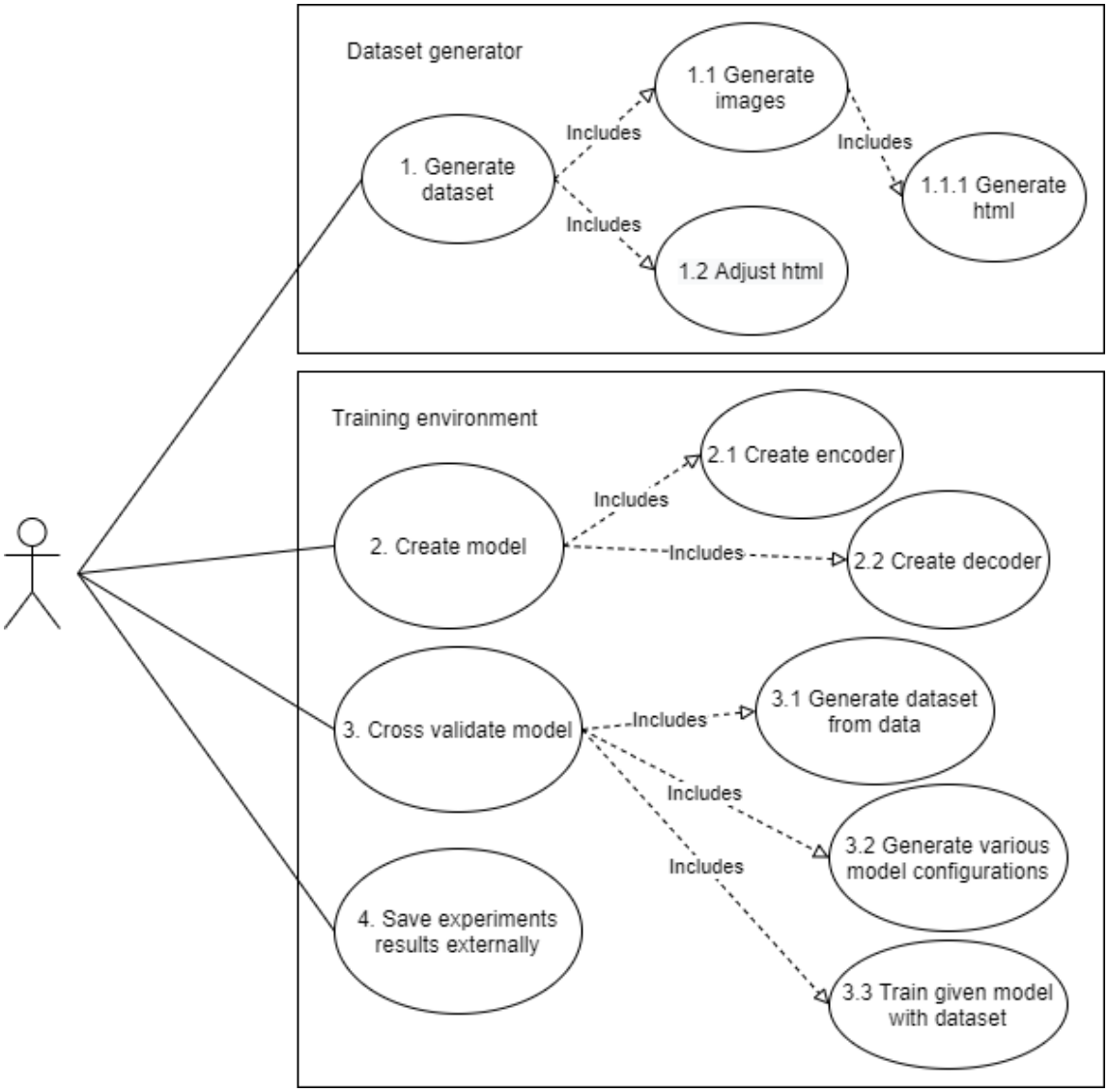

Fig. 24 Use case diagram of our solution

Table 4 Dataset generation use case

Use case id	1
Name	Generate dataset
Description	Generates multiple HTML documents that would follow a given set of rules.
Actors	System
Initial requirements	The set of rules must be provided
Main steps	1.1.1. Take the set of rules, creates HTML code from it, and saves it into multiple files.

	1.1.2. Process files generated in 1.1.1 by puppeteer resulting in webpage screenshots. Finally, all results are shuffled and split into subsets. 1.1.3. Html is adjusted by replacing not Html tokens with an unknown token. Available Html tokens are written to JSON file.
Final results	A new dataset is split into the test, and train/validate subsets, each containing a JSON file with an array of sequences and image names and a list of image files.

Table 5 Model creation usecase

Use case id	2
Name	Create model
Description	Creates a model
Actors	System
Initial requirements	Model and encoder type must be known
Main steps	2.1 Create encoder 2.2 Created decoder
Final results	Funcional keras model

Table 6 K-fold cross-validation on the given model use case

Use case id	3
Name	Cross validate the model with data
Description	Splits dataset into k-folds, fits given model onto various data combinations calculate average
Actors	System
Initial requirements	Model, dataset and fold size must be provided
Main steps	3.1 Generate dataset from data 3.2 Generate various models configurations 3.3 Train give dataset with model
Final results	Fitted models and their training results

Table 7 Save experiments externally

Use case id	4
Name	Load and demonstrate the model
Description	Calculates prediction for specific token data and save experiments results externally
Actors	System
Initial requirements	Predictions and original dataset must be present
Main steps	Calculates top k 1 accuracy, calculate top k 5 accuracy, calculated unpredicted tokens ratio, calculated wrong prediction rate
Final results	Prediction metrics are save in weight and biases

2.1.Dataset generator

To compare our model result to Pix2code results first, we must synthesise the dataset in which images are similar to Pix2code while the sequence is entirely different.

2.1.1. Requirements for dataset synthesiser

Dataset generator requirements can are split into functional and non-functional requirements. They are listed below. Functional requirements:

- Generate multiple HTML documents that would follow a given set of rules.
- Generate an image from a given HTML document.
- Generate dataset, which later will be used for training and evaluating the model.

Non-functional requirements are:

- Ensure that rare samples are included in the dataset.
- It is implemented using ECMAScript-262.
- Use pseudo-random selection for sequence generation.
- Use Durstenfeld shuffle.

Generating a pix2code dataset is not a trivial task because that dataset has 10477080 unique possible variants from which we will be using only 1750 (1500 for training/validating and 250 for testing). Because we are following amounts used in pix2code. All possible dataset variants can be calculated using the formula (1):

$$n = h * \sum_{r=R} \sum_{c \in C} s^{c*r} \quad (1)$$

Where n is the total number of unique variants, h is the number of header combinations, R is the set of possible row amount (in our case, it is 1,2,3), C is the set of possible columns combinations (in our case, it is 1,2,4), and s is the number of possible standard block variants. Data entry can contain two up to 5 header buttons (buttons at the top of *Fig. 25 Training image sample*). The single one is blue, while the others are black, resulting in 5! = 120 possible combinations. The standard element is an element shown in a heading text, plain text, and a button, either red, green, or yellow, resulting in three unique standard blocks.

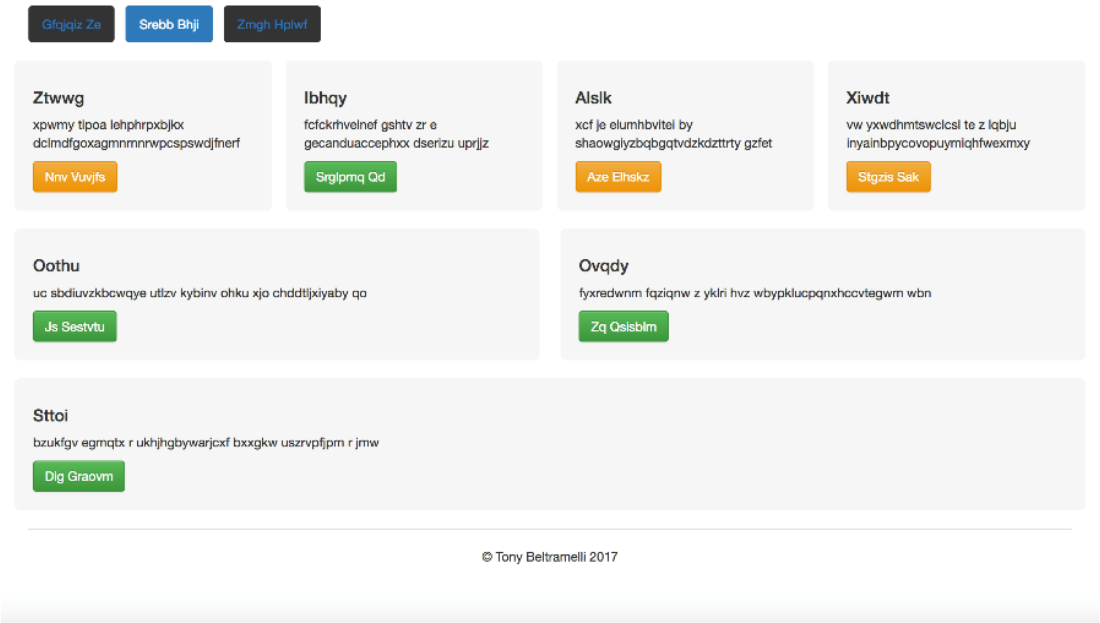

Fig. 25 Training image sample

Therefore, resulting in 10477080 variants, and since we are taking only 1500 out of them, the chance of our training dataset would include an entry that contains one single column with a single row is ~6.01%. That arises an issue that our model could be completely unaware of such an instance. We address it by splitting data into 38 sets where each entry has a unique sequence of standard block positioning and size. We take 54 cases from each group and randomly select two additional dataset entries.

2.1.2. Dataset synthesiser components overview

Dataset synthesiser can is split into four main components:

1. Variant's generator
2. Html generator
3. Image generator
4. Dataset collector

Dataset collector collects all the data in each directory by extracting HTML content and writing it to a single JSON file. Image generators iterate through HTML files, creating their respective image as a screenshot for each rendered HTML document and then resizes it to given dimensions. Since both the image generator and dataset collector use the file system as their inputs, they can generate a dataset from any Html files. The HTML generator creates HTML from a given tree structure. That created document is stored in a given directory. Once the files are stored, the variants generator calculates all possible variants from given parameters, selects the pseudo-random number of used data instances, and splits the dataset into train/validate and test datasets. We will be comparing results to pix2code. We need our dataset synthesis to replicate their dataset images. Integral components are displayed in *Fig. 26 Dataset synthesiser component diagram*.

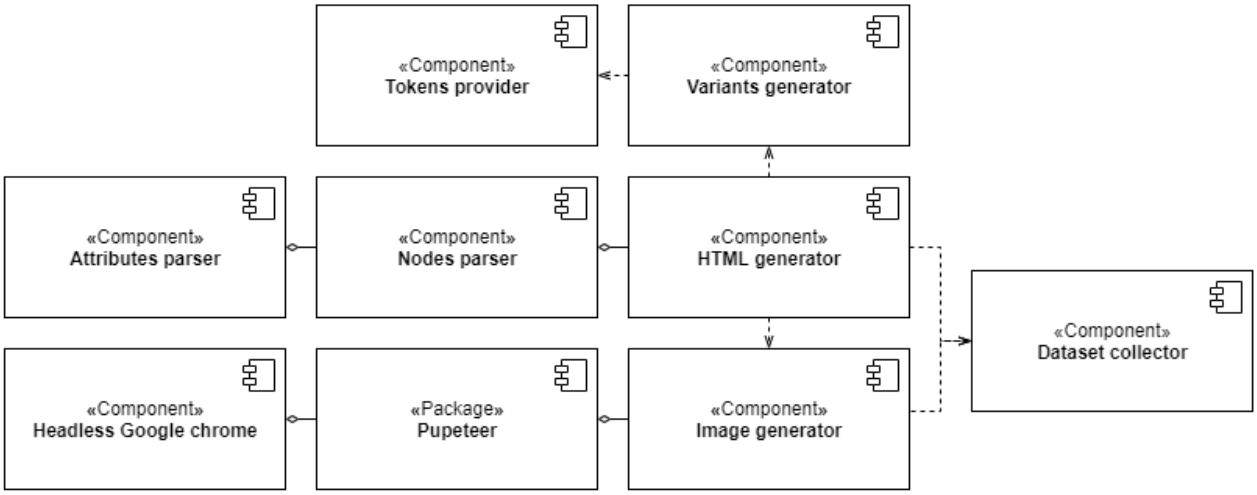

Fig. 26 Dataset synthesiser component diagram

2.2. Training environment

The training environment is where we execute all the experiments and collect all the data for our research.

2.2.1. Requirements for environment

Training environment requirements can be split into functional and non-functional requirements. They are listed below. Functional requirements:

- Able to train the model.
- Do K-cross validation.
- Provide training metric logs.
- Save metric logs to external storage.
- Demonstrate the model.

Non-functional requirements are:

- Contained inside Jupyter notebook format.
- Be computing provider environment is independent, meaning that the environment would work the same in Google collab, paper space, or Floyd hub environments.
- Utilise Weighs and Biasias.
- Be trainable on using TPU.

2.2.2. Training environment overview

Our training environment is made out of 4 main components: variants trainer, K-fold cross validator, Model creator, and Dataset loader. Detailed components diagram is shown in *Fig. 27 Training environment component overview.*

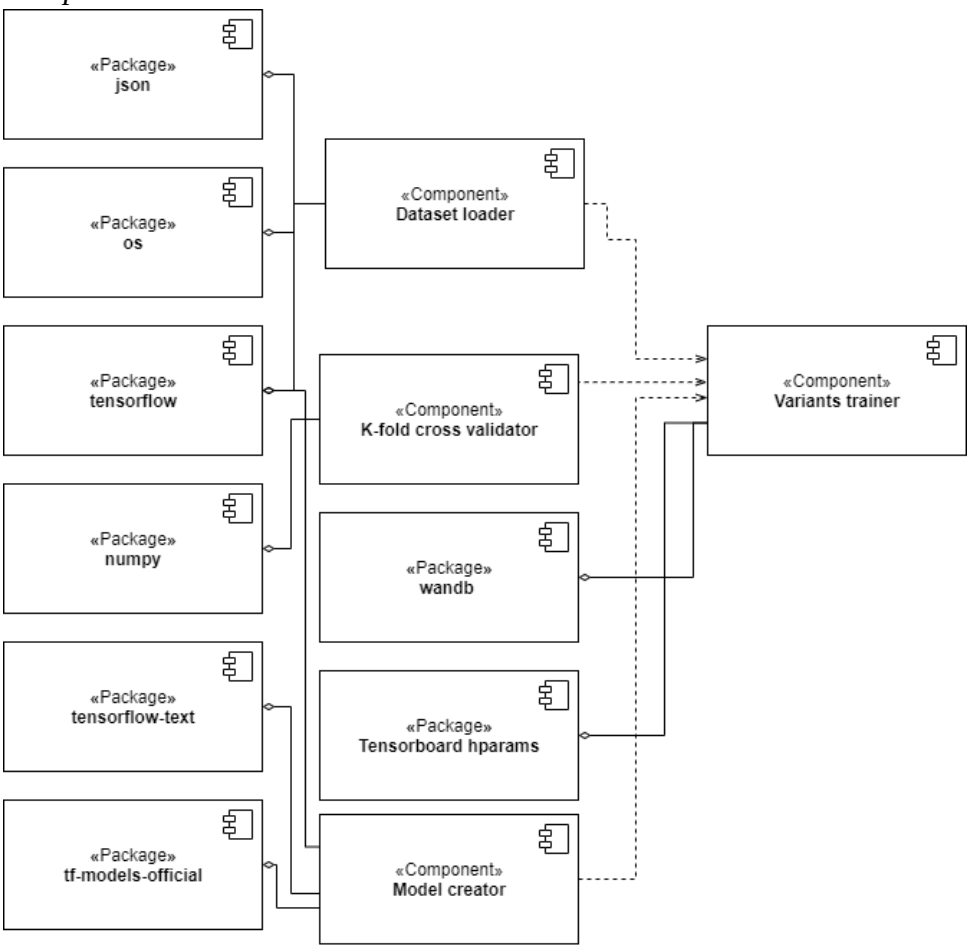

Fig. 27 Training environment component overview

The variants trainer using the hparams package modifies various hyperparameters for models provided by the model creator, which is then passed to the k-fold cross validator and the dataset loader's data. K-fold cross validator trains the model and returns every fold results and the accumulated result to variants trainer, which then passes this data to the wandb package to be saved externally inside weight and biases platform.

2.2.3. The optimiser used for model training

While training our model for weight optimisation, we will be using Adam optimiser with weight decay because Adam L2 regularisation is not practical compared to standard gradient descent (30). Having L2 regularisation decreases the chance to overfit our model. Our optimiser also has to have warmup steps because having them is shown to benefit attention mechanism based architecture accuracy. Our exact learning rate is shown in *Fig. 28 learning rate graph*.

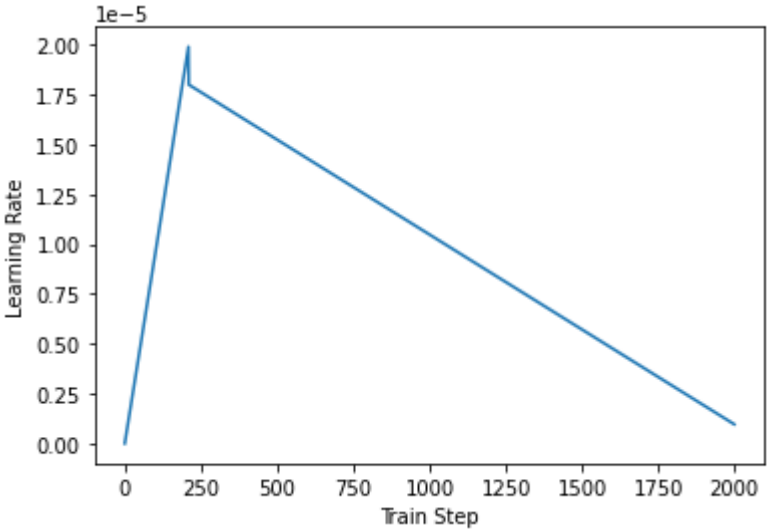

Fig. 28 learning rate graph

2.2.4. K-fold cross-validation

K-fold cross-validation is a process that uses the same dataset by resampling the dataset into various compositions same size, training, and validation datasets. Pix2ode and Pix2html use the same size (1750) dataset, from which 250 samples are used for validation and the rest for training. Thus every different model configuration will be trained and evaluated seven more times. This strategy reduces the dataset entries' order impact on training and validating accuracy.

2.2.5. The model

Our model generator has requirements that can be split into functional and non-functional they are listed below. Functional:

- Generate models from a given set of parameters.
- Generated models could take an image testing given jpeg image.
- Model weights can be saved and loaded.
- Model does not run into exploding/diminishing gradient problem.

Non-functional:

- Reach better accuracy than pix2code.
- Use Keras functional API.
- Follow encoder-decoder architecture.
- Not consume more 16Gb of memory during training.
- Be trainable using TPU.
- Use transformer for the decoder.

2.2.6. Teacher forcing

Our model uses teache. Initially, the model would use a previously predicted token. However, when using teacher forcing, previous token actual sequences are provided instead of predicted ones. Thus the model can make the following prediction on correct data even if the previous prediction was wrong. To achieve this, we are passing the look ahead mask, which prevents the model from looking into future predictions.

2.2.7. Dropout

Our model should also include dropout layers. Drop is one of regularisation method which prevents the model from overfitting. The co-adaptation of cells can cause overfitting. However, this co-adaptation is broken by introducing a dropout (making various cells unavailable during the training period). Thus, cells no longer strongly rely on their specific previous layer cells and pay more attention to overall input. That enables the model to generalise more accurately. (31)

2.2.8. Model layers overview

Our model follows encoder-decoder architecture where the decoder is a Transformer that uses information from both: available sequence and image. As for the encoder, we are using either CNN or an image processing transformer. Thus our network looks like a Transformer with stacked attention modules architecture as shown in *Fig. 29 Captioning transformer with stacked attention modules.* Or as full transformer architecture image captioning demonstrated in ***Fig.** 21 Full transformer for image captioning network visualised*

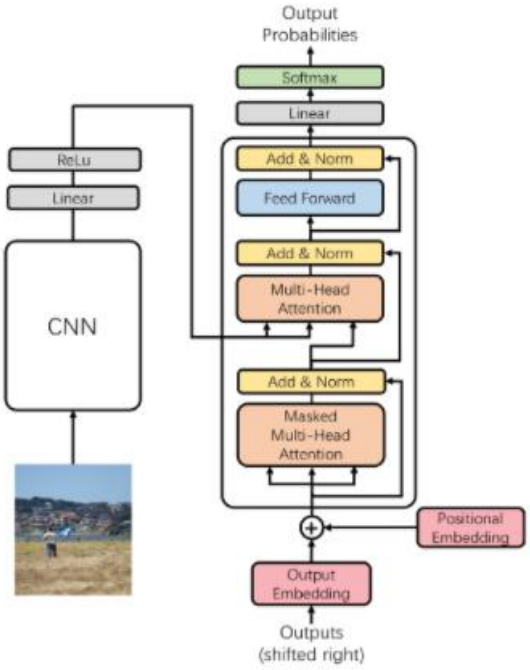

Fig. 29 Captioning transformer with stacked attention modules

2.3. Data management for our solution

Since the local environment is not an option for cloud providers, we will store our data in various data storage hardware and software solutions; thus, our components can be stored in multiple platforms communicate via TPC/IP protocol as shown in *Fig. 30 Our solution deployment diagram.*

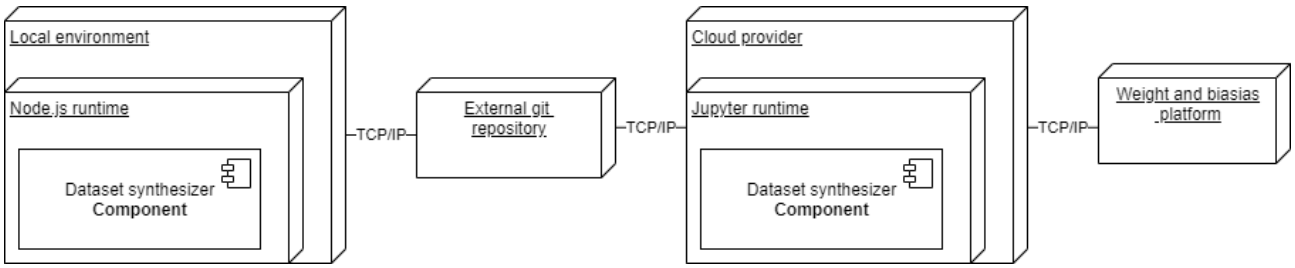

Fig. 30 Our solution deployment diagram

Dataset will store in the git repository, and experiment results will be in the weights and biases platform. Keeping data this way will enable us to recover quickly from various crashes and return experiments if they yield some unexpected results to double-check. Finally, by saving our dataset in weights and biases platform, we should take advantage of various visualisations and insights provided by this platform.

3. Implementation

In this section, we will overview the implementation details of our dataset synthesiser and our training environment.

3.1.1. Dataset synthesiser

We are using NodeJs 15.5.0 as a runtime to perform dataset synthesis. Datasets synthesiser is written using ECMAScript because of many helpful and up-to-date packages for generating images from HTML. NodeJs (ECMAScript runtime) can also be easily installed on many operating systems. We aim to achieve images similar to pix2code, allowing our model to gain a decent result with pix2code images while training our model with our dataset. Original pix2code image on the left, and our dataset image is on the right.

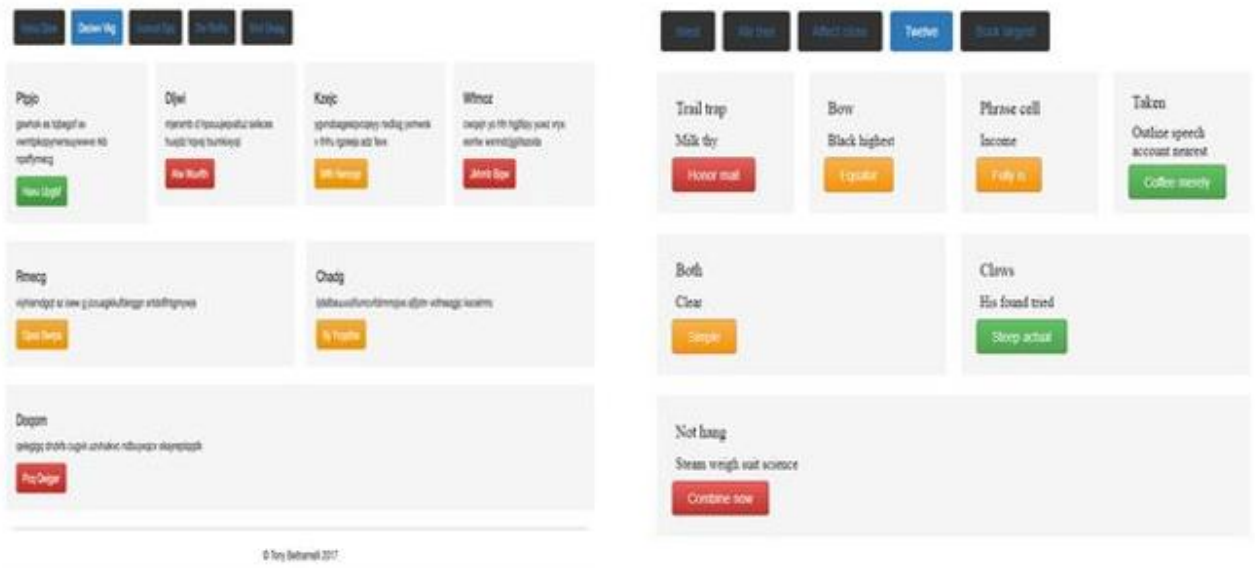

Fig. 31 Pix2code and our dataset images

Our dataset sequence is generated from 85 unique tokens, while the pix2code dataset has 16 unique tokens. Our max sequence length is 1827, while the pix2code dataset max sequence length is 88. We generate 1750 dataset entries to match the pix2code dataset amount. We split our dataset into 1500 training and 250 evaluating instance pairs. Split is done by firstly shuffling the dataset using Durstenfeld shuffle and taking the first 1500 pairs to train, while the rest goes to the test dataset. Comparison between our is shown in table *Table 8 Comparison between pix2code and pix2html dataset*.

Table 8 Comparison between pix2code and pix2html dataset

Dataset	Unique tokens	min length	Average length	Max
Pix2code	16	16	~53.95	90
Our	73	132	~481.31	860

Detailed examples of pix2code and our dataset entry contained a header with five buttons and a single row single block containing a text is given below

```
     header {
        btn-inactive ,
        btn-inactive ,
        btn-inactive ,
        btn-active ,
        btn-inactive
     }

     row {
        single {
           small-title,
           text,
           btn-red
        }
     }
```

```
<body>
<header style="display:flex; flex-direction:row; margin:15px 0; ">
<button style="background:#333333; color:#2f79b9; margin:5px; padding: 0 20px; align-self:baseline;
border-radius:4px; height:40px; font-size:14px; border:none; "> Sent stiff </button>
<button style=" background : #333333 ; color : #2f79b9 ; margin : 5px ; padding : 0 20px ; align-
self : baseline ; border-radius : 4px ; height : 40px ; font-size : 14px ; border : none ; "> Harder
</button>
<button style=" background : #333333 ; color : #2f79b9 ; margin : 5px ; padding : 0 20px ; align-
self : baseline ; border-radius : 4px ; height : 40px ; font-size : 14px ; border : none ; "> Upward
list </button>
<button style=" background : #2f79b9 ; color : #ffffff ; margin : 5px ; padding-right : 20px ;
padding-left : 20px ; align-self : baseline ; border-radius : 4px ; font-size : 14px ; height : 40px
; border : none ; "> Stems traffic </button>
<button style=" background : #333333 ; color : #2f79b9 ; margin : 5px ; padding : 0 20px ; align-
self : baseline ; border-radius : 4px ; height : 40px ; font-size : 14px ; border : none ; ">
Receive toy </button>
</header>
<div style=" display : grid ; grid-template-columns : repeat(4,1fr) ; gap : 20px ; grid-template-
rows : repeat(3,140px) ; ">
<div style=" background-color : #f5f5f5 ; grid-row : 0 ; border-radius : 4px ; padding : 20px ;
display : flex ; flex-direction : column ; justify-content : space-around ; grid-column : 1/5 ; ">
<h4 style=" margin : 5px ; font-size : 18px ; font-weight : 500 ; "> Inside buy </h4>
<span style=" margin : 5px ; "> Leave church capital </span> <button style=" color : white ;
background-image : linear-gradient(#fbb450,#f89406) ; padding-right : 20px ; padding-left : 20px ;
align-self : baseline ; border-radius : 4px ; border-color : rgba(0,0,0,0.25) ; border-style : solid
; border-width : 1px ; min-height : 34px ; font-size : 14px ; text-shadow : 0 -1px 0 rgba(0,0,0,0.2)
; box-shadow : inset 0 1px 0 rgba(255,255,255,0.15), 0 1px 1px rgba(0,0,0,0.08) ; "> Brought numeral
</button>
</div>
</div>
</body>
```

3.1.2. Implementation of model

We have implemented four different models. All of them follow the encoder-decoder structure. Three of them uses cnn for the encoder, while the other uses a transformer. For the decoder, we use a transformer like a decoder with two-layer decoder layers. Implemented architectures loosely follow image captioning with stacked attention. However, we do not calculate the combined loss. We have tried using Relu as in the original transformer, and we have also tried Gelu because it is used in training state of the art models such as GPT-2 and Bert. We found that using Gelu yields better results. We do use Adam with a weight decay optimiser with a custom learning rate that includes warmup steps. All of the models implemented using Keras functional API and TensorFlow and TF-models package. Using this technology stack enables us to use nonlinear topology models with multiple inputs and efficiently scale on multiple GPU and even use TPU pods. Our model accepts the following hyperparameters: model size, number of attention heads, vocabulary and dropout size. Both CNN and Transformer encoders based models are shown in *Fig. 32 Our models' implementation*

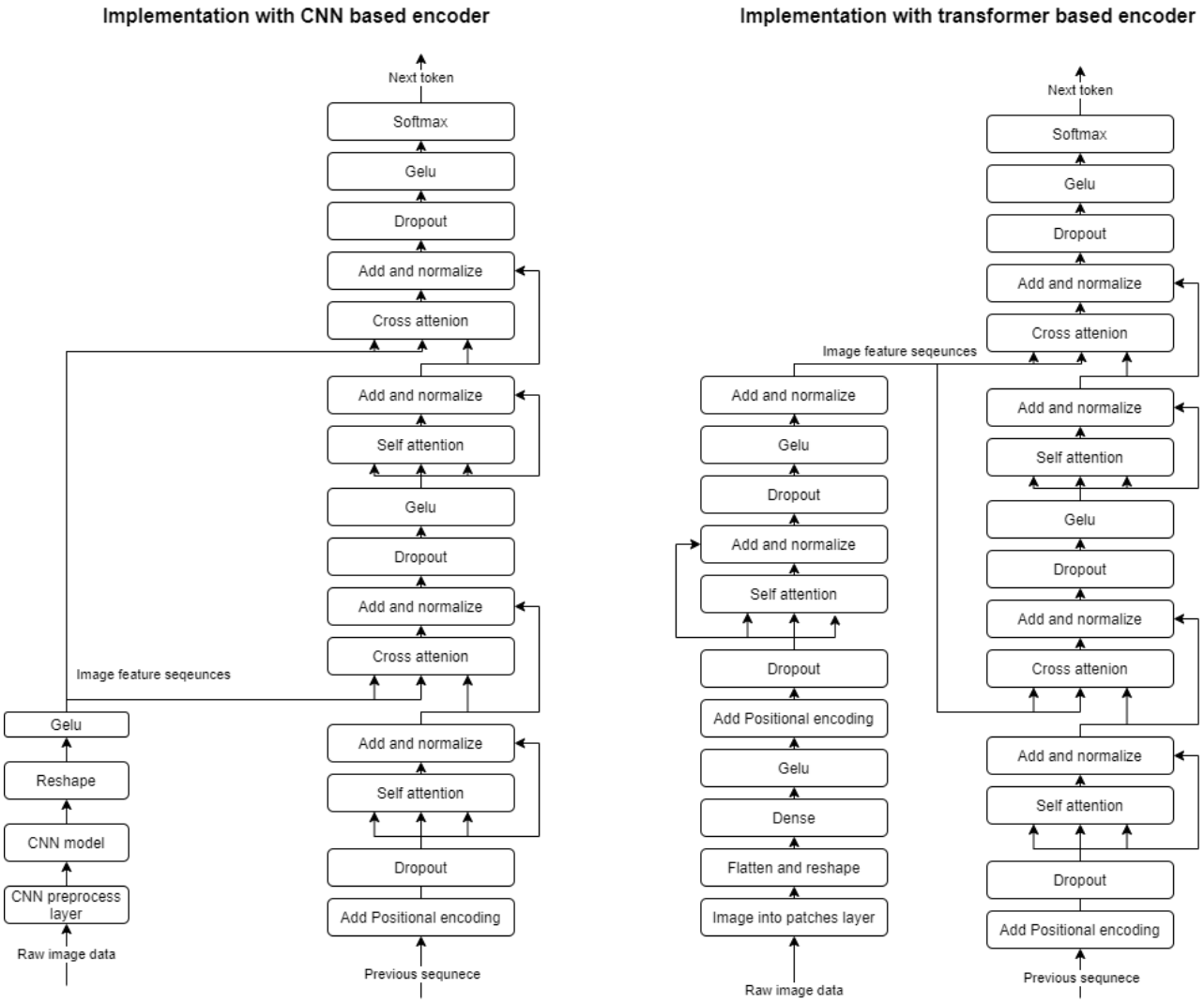

Fig. 32 Our models' implementation

3.1.3. Implementation of training environment

Our training environment is implemented using Jupyter notebook, tensorboard hparams API, and weight and biases. We use hparams to set a list of hyperparameters to do a grid search for optimal parameters and weight and biases for saving and visualising training logs. The training environment also takes care of loading and preprocessing data later used for model training. We sort both (training and validation) datasets by tokenised sequence length, which improves model training performance. Then we tokenise HTML using tokens provided by the dataset synthesiser. After tokenisation, we pad our text sequences with padding tokens, making each sequence vector the same length. Lastly, we preprocess images, turning their web page screenshots into tensors, first by resizing them to 224x224 dimensions and then normalising each pixel value. Once our HTML and image are turned into a sequence, they are passed to the model fit. While searching for optimal hyperparameters, we used k-fold cross-validation. We split our dataset into two parts: test and training datasets. We do not use testing dataset for hyperparameters search. Then we split our training dataset into six parts, each containing 250 entries. Five of them make a dataset on which models are trained, and one is used to calculate validation accuracy. Then we rearrange a set by moving the set used for validation into the training dataset and choosing the not already used set for validation. We repeat this process six times, and we calculated all the metrics by adding all results and dividing them by the number of folds. We train our models for 30 epochs using a batch size of 25

4. Experiments

In this section, we overview our experiments in which we are evaluating four different models based on two different architectures,

4.1. Preparation for experiments

In this subsection, we describe every experiment steps in details. The raw dataset is processed, then the model is constructed, then it is then evaluated with train/validation with processed dataset using k-fold cross-validation. Finally, each most accurate configuration model is then trained again using the whole train/validate dataset for training and evaluated using the test dataset.

4.1.1. Dataset processing

Both datasets have only known tokens during the data preprocessing stage. We use a batch size of 25. Using a larger batch size would result in more than 16GB usage of random access memory for the full transformer architecture model above our non-functional requirements.

4.1.2. Models

Three of them are following image captioning with stacked attention and using processing. While every model decoder uses transformers like architecture, they have different encoders. We have chosen three CNN architectures because their input dimensions are 224x224, and they are already available as a part of the Keras library. We also implemented vision as a transformer as an encoder for the fourth model. We have chosen a hidden layer size of 120 because it second least common denominator of numbers from one to six (a number of heads used for hyperparameters optimisation). We also have a tried models with a hidden layer size of 60. However, it resulted in very low accuracy with our dataset. Detailed encoder comparison is shown in *Table 9 Encoders comparison*.

Table 9 Encoders comparison

Encoder	Input pixel size	Trainable params	Non-trainable params
Vgg16	224x224	14,776,248	0
resnet	224x224	23,765,240	45,440
EfficientNetB0	224x224	4,161,268	42,023
Vit16	224x224	469,680	0

4.1.3. Model evaluation

Firstly, we did various models grid search for both datasets. We use k fold cross-validation with six-folds. Each fold is trained for 30 epoch with a batch size of 25 and the optimiser described in 2.2.3. We evaluated the model's accuracy from one to six attention heads and three different dropout values. Since we have two different datasets and four different models, this resulted in 144 different combinations being evaluated using k-fold cross-entropy. Hyperparameters used for grid search are shown in *Table 10 Hyperparameters used for grid search*. We calculate their average train and validate accuracies. Thus we can compare how dropout and the number of attention of heads impact overall results and overfitting.

Table 10 Hyperparameters used for grid search

Hyperparameter name	**Possible value**
Attention heads used in multi-head attention	6,5,4,3,2,1
Dropout	0.0, 0.05, 0.1

After grid search, we take every type of model best-performing variant, train it on a train/validate dataset and evaluate it on the test dataset, and their results analyse the following metrics: train accuracy, validate accuracy, confusion matrix and accuracy report (precision, recall, f1-score and support)

4.2. Experiments with pix2code dataset and model with VGG16 encoder

This section will overview grid search results with our models and pix2code dataset and analyse the best performing model accuracy.

4.2.1. Grid search results

The best performing model had 0.05 dropout and six attention heads and achieved 0.9404 validation and 0.9428 training accuracy. Also, it is worth noticing that models with a 0.05 dropout achieved a better average than models without a dropout. Models with four attention heads achieved slightly better average validation accuracy (0.9333) than models with five heads (0.9325).

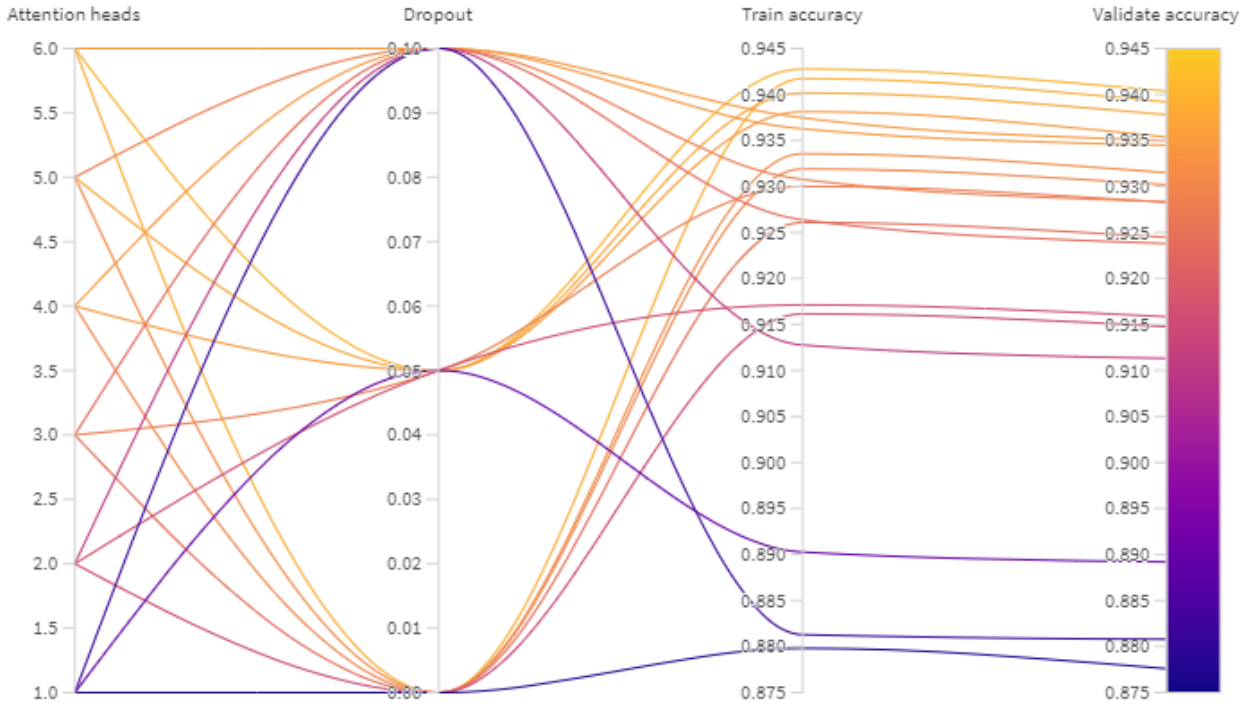

Fig. 33 Model with VGG16 grid search results

4.2.2. Best performing model confusion matrix and classification report

According *Table 11 Model with VGG16 encoder accuracy report* model achieves one F1 score on the whole header row. It has no problem correctly predicting the whole header row, including inside buttons and opening-closing tokens. The lowest F1 score is for button green, orange and red buttons (0.49, 0.46 and 0.36).

Table 11 Model with VGG16 encoder accuracy report

Label	**Precision**	**Recall**	**F1-Score**	**Support**
<pad>	1	1	1	9097
Header	1	1	1	250
{	1	1	1	2462
Btn-active	1	1	1	250
Btn-inactive	1	1	1	629
}	0.99	0.99	0.99	2462
row	0.99	0.99	0.99	666
single	0.99	0.96	0.98	228
small-title	0.99	0.96	0.98	1546
text	1	1	1	1546
btn-orange	0.47	0.46	0.46	508
double	0.98	0.04	0.98	434
Btn-red	0.36	0.36	0.36	514
quardruple	0.99	0.99	0.99	884
Btn-green	0.47	0.51	0.49	524
start	0.92	0.96	0.94	250

According to Fig. 34 Model with VGG16 encoder, confusion matrix Model is often confusing red, orange and green buttons.

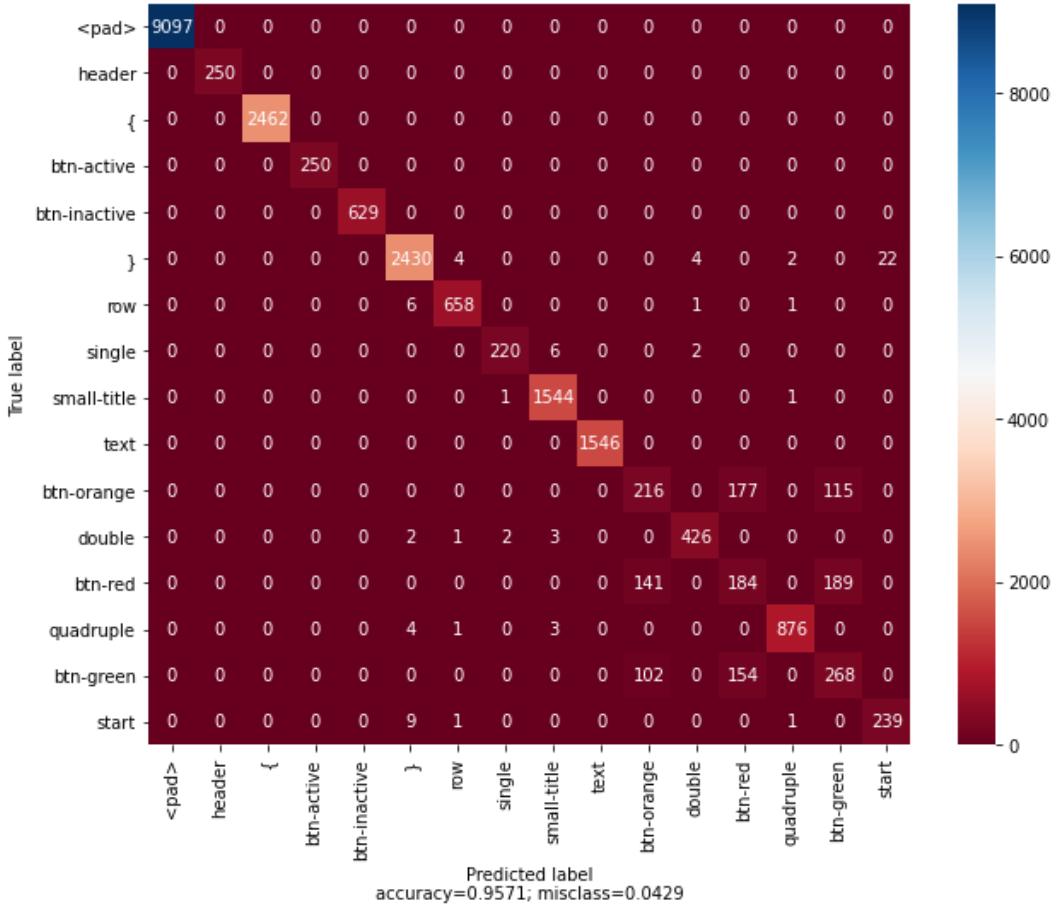

Fig. 34 Model with VGG16 encoder confusion matrix

4.3. Experiments with pix2code dataset and model with Visual transformer encoder

This section will overview grid search results with our models and pix2code dataset and analyse the most accurate model with a visual transformer encoder.

4.3.1. Grid search results

Best results were achieved with six heads and 0 dropout (0.8633). However, the configuration with five attention heads reached similar accuracy (0.861). The average five attention heads accuracy is 0.8544, while configurations with six heads have average validation accuracy of 0.8565. This indicated that increasing attention heads from five to six attention heads does not significantly increase. Also, all zero dropout configurations outperformed 0.05 dropout configurations with the same attention heads count. Furthermore, 0.05 configurations outperformed configurations with 0.1 dropout, while having dropout overfitting (different between training and validation accuracy) is decremental to overall validation accuracy.

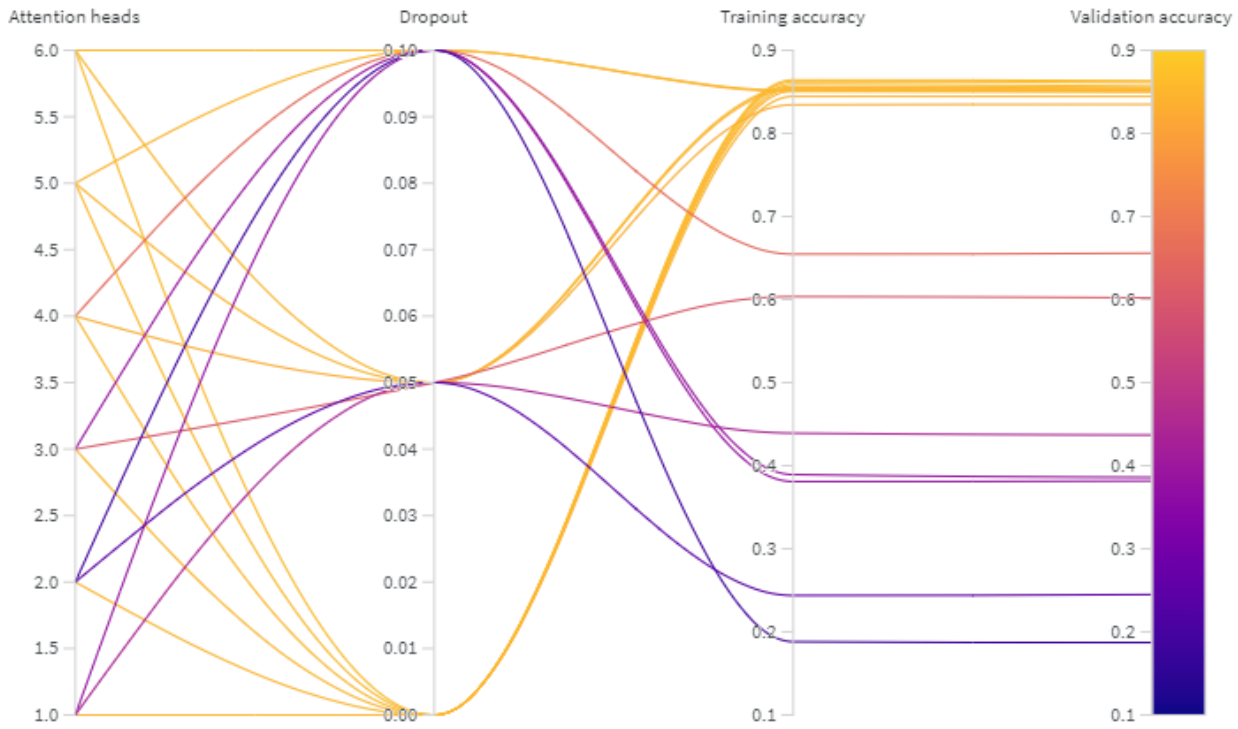

Fig. 35 Model with visual transformer encoder grid search results

4.3.2. Most accurate model confusion matrix and classification report

According to the *Table 11 Model with VGG16 encoder accuracy report*, the first two tokens of a sequence (Header and {) got have F1 score of 1 and 0.99. Padding token and text token also have a precision score of 1. Token of double and single have a relatively low F1 score (0.15 and 0.21). All three (orange, red and green) buttons have similar F1 scores (0.42, 0.46 and 0.48 and precisely the same precision of 0.45.).

Table 12 Model with Visual transformer encoder accuracy report

Label	**Precision**	**Recall**	**F1-Score**	**Support**
<pad>	1	1	1	9097
Header	1	1	1	250
{	1	0.99	0.99	2462
Btn-active	0.36	0.068	0.11	250
Btn-inactive	0.64	0.83	0.72	629
}	0.82	0.86	0.84	2462
row	0.64	0.62	0.63	666
single	0.46	0.14	0.21	228

small-title	0.75	0.99	0.86	1546
text	1	1	1	1546
btn-orange	0.45	0.46	0.46	508
double	0.41	0.094	0.15	434
Btn-red	0.45	0.39	0.42	514
quardruple	0.56	0.53	0.55	884
Btn-green	0.45	0.5	0.48	524
start	0.77	0.71	0.74	250

According to *Fig. 34 Model with VGG16 encoder confusion matrix* this model confuses orange, red and green buttons and between active and inactive buttons. It also mixes double, quadruple and } tokens.

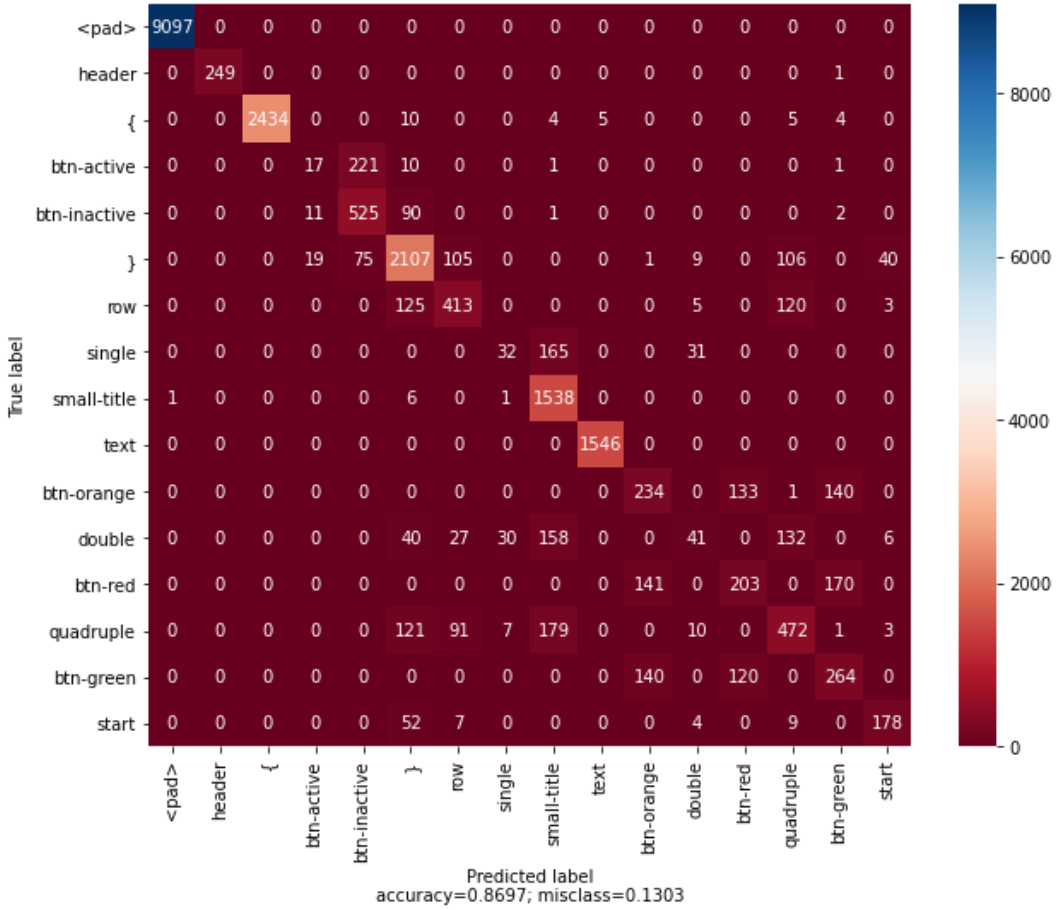

Fig. 36 Model with Visual transformer encoder confusion matrix

4.4. Experiments with pix2code dataset and model with EfiecientNet encoder

This section will overview grid search results with our models and pix2code dataset and analyse the best performing model accuracy.

4.4.1. Grid search results

According to *Fig. 37 Model with EfiecientNet encoder grid* search best performing model had six attention heads and 0 dropout. Also, all three best performing models have six attention heads. Thus we can reason that EfficientNet provides features that can benefit from additional dimensions in multimodal space. Additionally, increasing dropout did not significantly reduce model overfitting.

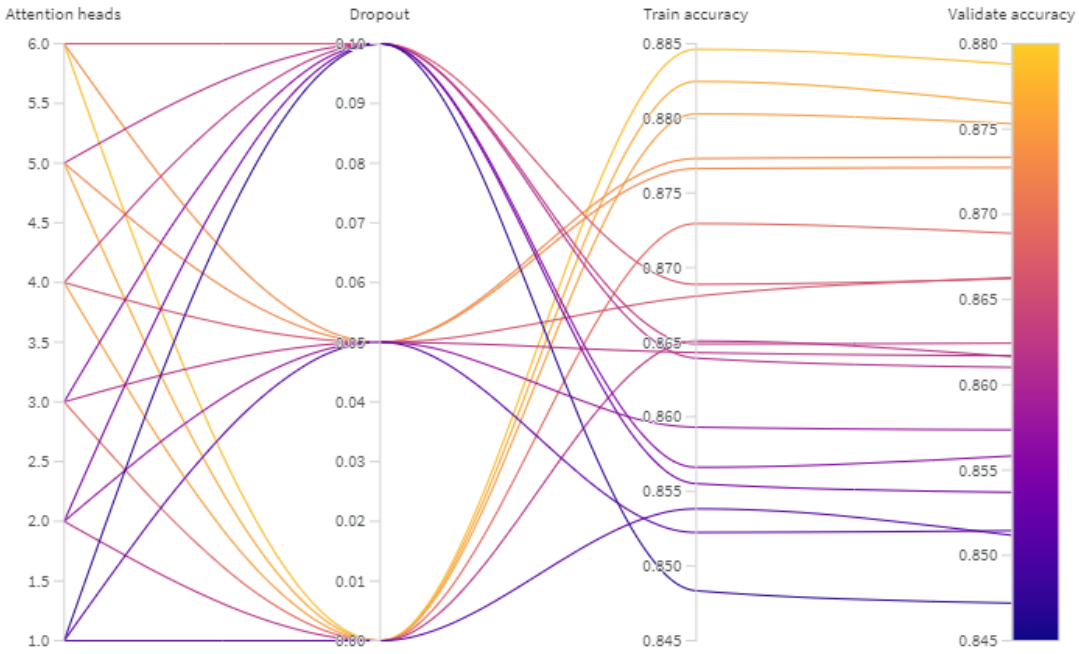

Fig. 37 Model with EfiecientNet encoder grid search

4.4.2. Best performing model confusion matrix and classification report

Similarly to the VGG16 and visual transformer, it has one F1 score first tokens (Header and {) as well padding token and text token. It struggles with orange, red and green buttons and active Whole accuracy report, shown in *Table 13 Model with EfiecientNet accuracy report*.

Table 13 Model with EfiecientNet accuracy report

Label	**Precision**	**Recall**	**F1-Score**	**Support**
<pad>	1	1	1	9097
Header	1	1	1	250
{	1	1	1	2462
Btn-active	0.59	0.26	0.36	250
Btn-inactive	0.75	0.8	0.77	629

}	0.86	0.88	0.87	2462
row	0.74	0.73	0.74	666
single	0.68	0.61	0.64	228
small-title	0.85	0.97	0.91	1546
text	1	1	1	1546
btn-orange	0.33	0.45	0.38	508
double	0.56	0.48	0.52	434
Btn-red	0.3	0.15	0.2	514
quardruple	0.88	0.73	0.8	884
Btn-green	0.35	0.41	0.38	524
start	0.57	0.78	0.7	250

According *to Fig. 38 Model with EfiecientNet confusion matrix*, this model mixes coloured button predictions and double with quadruple and single tokens.

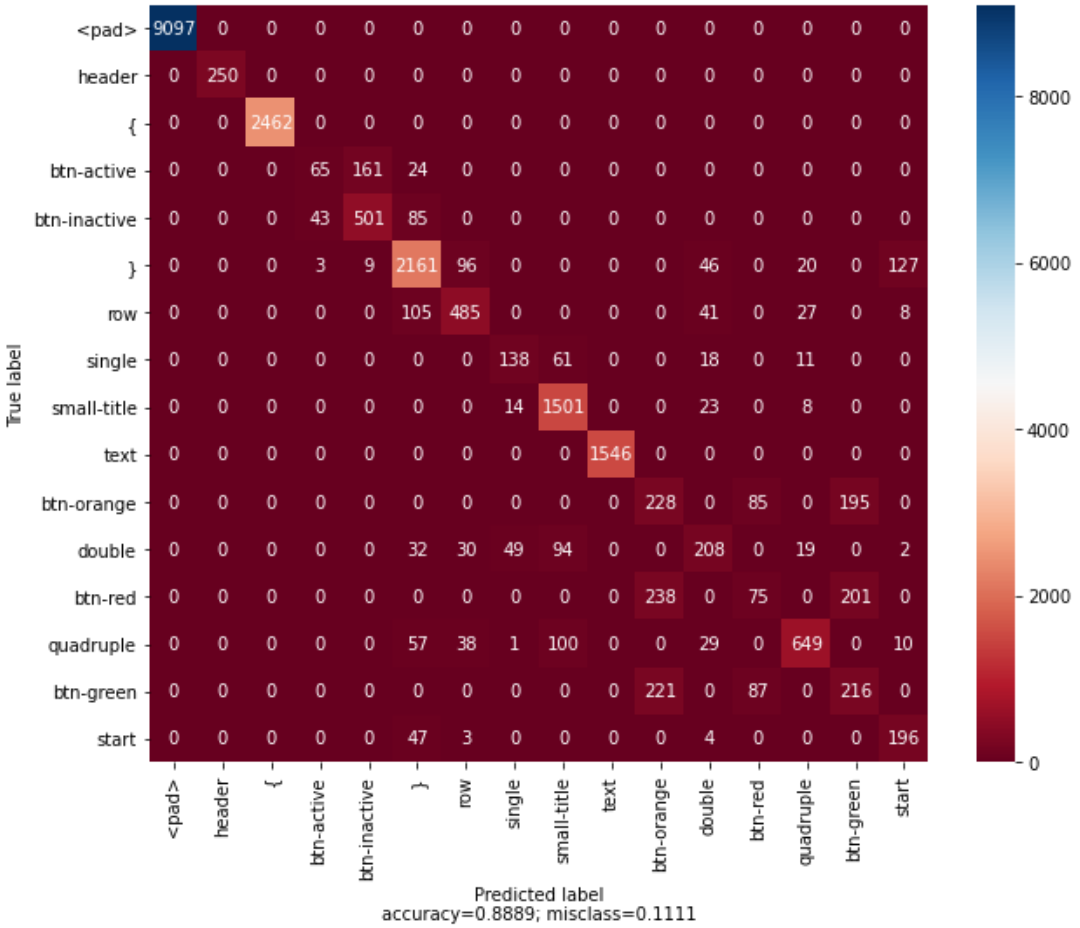

Fig. 38 Model with EfiecientNet confusion matrix

4.5. Experiments with pix2code dataset and model with ResNet encoder

This section will overview grid search results with our models and pix2code dataset and analyse the best performing model accuracy.

4.5.1. Grid search results

According to *Fig. 39 Model with Resnet encoder grid search results*, the best resulting combination is six attention heads and 0.05 dropout. It reaches 0.9303 validation accuracy. Interestingly configurations with 0.05 dropout on average outperform their counterparts with different dropout.

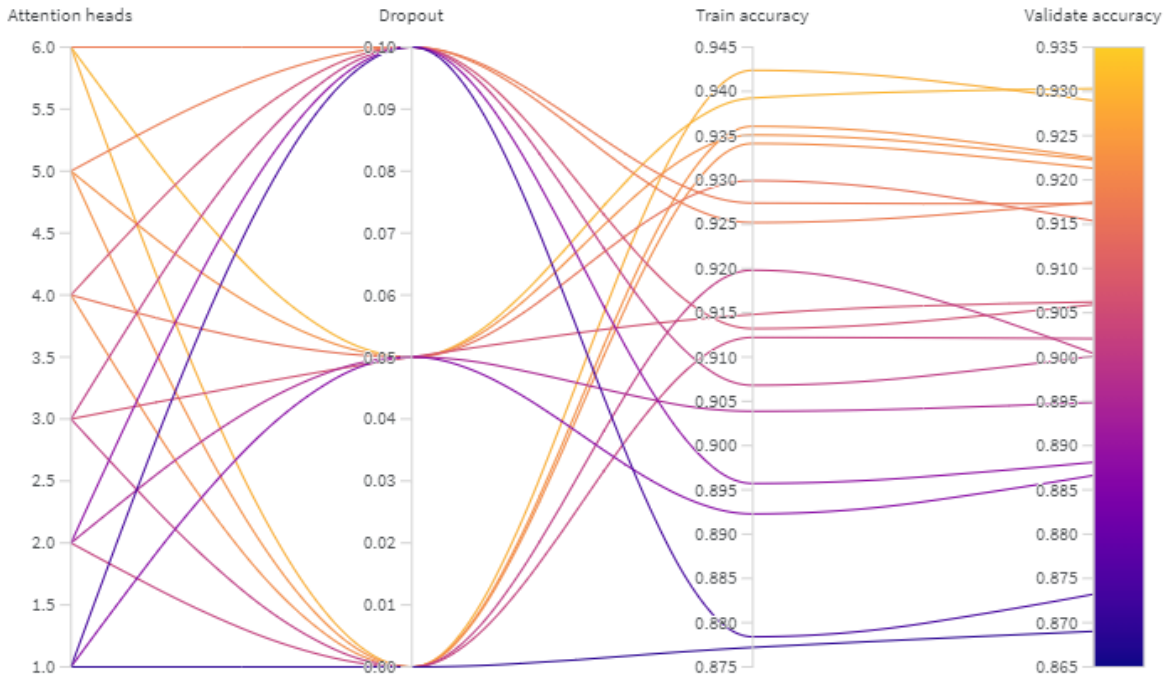

Fig. 39 Model with Resnet encoder grid search results

4.5.2. Best performing model confusion matrix and classification report

According to *Table 14 Model with Resnet encoder accuracy report* this configuration also predict the whole header row precisely. It also achieves the lowest F1 score with red, green and orange buttons.

Table 14 Model with Resnet encoder accuracy report

Label	Precision	Recall	F1-Score	Support
<pad>	1	1	1	9097
Header	1	1	1	250
{	1	1	1	2462
Btn-active	1	1	1	250
Btn-inactive	1	1	1	629
}	0.94	0.97	0.96	2462
row	0.96	0.92	0.94	666
single	0.88	0.87	0.88	228
small-title	0.92	1	0.95	1546

text	1	1	1	1546
btn-orange	0.43	0.38	0.41	508
double	0.85	0.53	0.65	434
Btn-red	0.38	0.45	0.42	514
quardruple	0.87	0.87	0.87	884
Btn-green	0.41	0.38	0.39	524
start	0.94	0.83	0.83	250

According to *Fig. 40 Model with Resnet encoder confusion matrix* this model confuses reg, green and orange buttons, double and quadruple tokens.

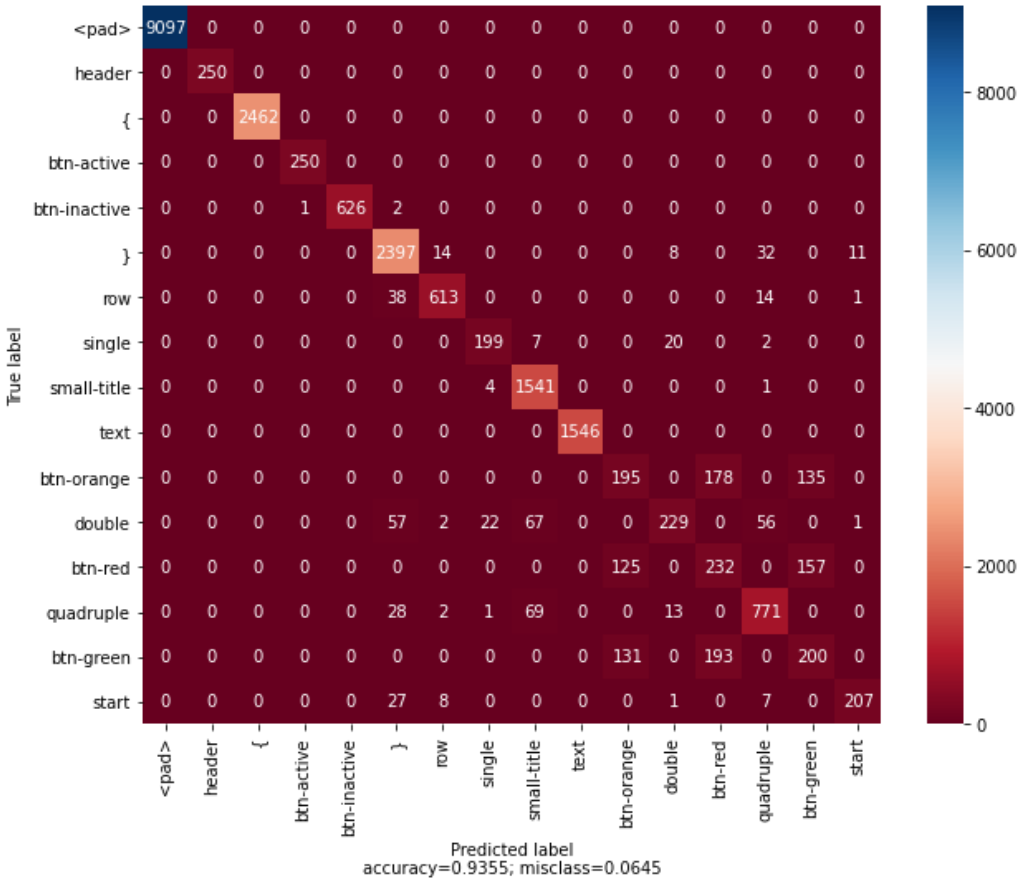

Fig. 40 Model with Resnet encoder confusion matrix

4.6. Experiments with pix2code overview

According to *Table 15 Each network top configuration with pix2code dataset results while doing a grid* search all the most accurate models have six attention heads. We can reason that having more attention heads enable extracting information from multiple dimensions in multimodal space. Best performing models (VGG16 and Resnet based encoders) have 0.05 dropout showing that they start to overfit, where training accuracy increase while validation accuracy drops.

Table 15 Each network top configuration with pix2code dataset results while doing a grid search

Network encoder	**Number of heads**	**Dropout**	**Validate Accuracy**	**Test accuracy**
VGG16	6	0.05	0.9428	0.9404

ResNet	6	0.05	0.9393	0.9303
Vit16	6	0	0.8643	0.8633
EffNetB0	6	0	0.8846	0.8788

When comparing their F1 score to respective tokens, we can see that all of them struggle with button colours. While the visual transformer did not achieve the best overall accuracy, it made the best predictions regarding the button colours. Also, we can reason that our decoder learns a comprehensive set of recurring sequence patterns with placeholders for tokens that are likely to change and then tries to predict that token.

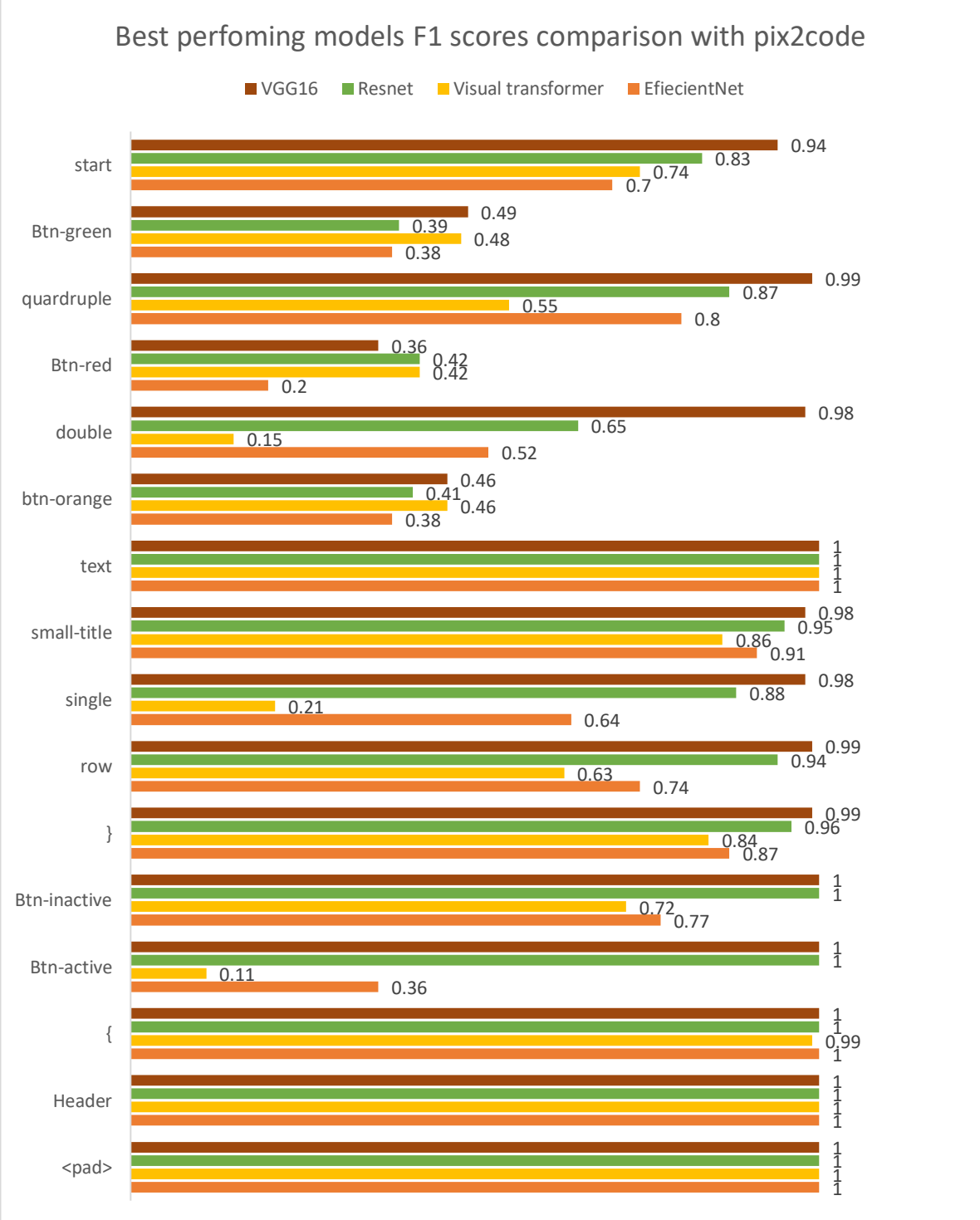

Fig. 41 Various models with pix2code F1 scores comparison

4.7. Experiments with our dataset and model with VGG16 encoder

This section will overview grid search results with our models and pix2code dataset and analyse best performing model accuracy.

4.7.1. Grid search results

According to *Fig. 42 Model with VGG16 encoder grid search results* best-performing model has a 0.05 dropout and five attention heads. Interestingly on average, zero dropout configuration were less accurate than configurations with 0.05 and 0.1 drop.

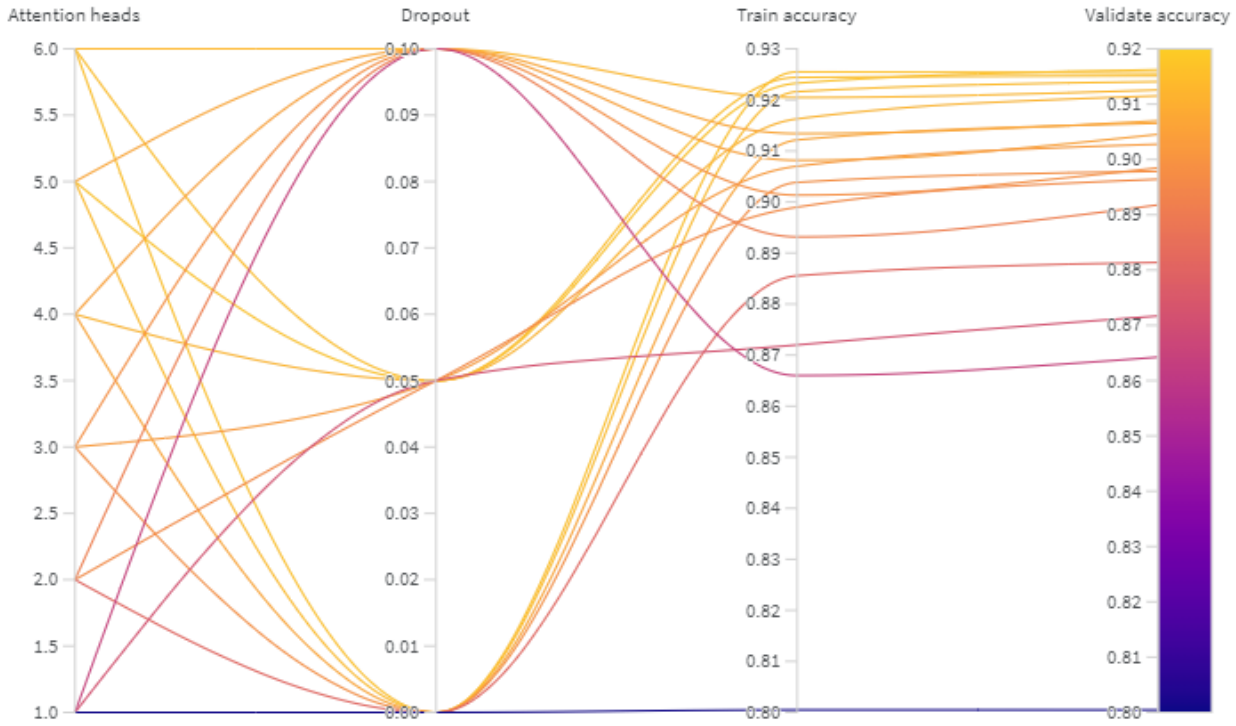

Fig. 42 Model with VGG16 encoder grid search results

4.7.2. Best performing model confusion matrix and classification report

According to *Table 25 Model with VGG16 encoder and our dataset precision report* and *Fig. 46 Model with VGG16 encoder and our dataset confusion matrix* we see that model usually misses prediction where similarly sequence has branches. Such as tokens ‘1/5’, ‘1/3’, and ‘3/5’, which always follow token ‘grid-column’. Similarly to pix2code dataset, it also mixes button colour defining tokens Model also confuses tokens such as ‘14px’ and ‘18px’, which always appear after token ‘font-size’. Also heardest to predict was a token with its value ‘3’ token ‘#ffffff’ value. ‘#ffffff’ is used to indicate white text colour for active header button element as well as button colour identifying tokens. Finally, it is the only model configuration that received a zero F1 score for any tokens.

4.8. Experiments with our dataset and model with visual transformer encoder

This section will overview grid search results with our models and pix2code dataset and analyse the best performing model accuracy.

4.8.1. Grid search results

According to ***Error! Reference source not found.***, the best validation accuracy was achieved by configuration of six attention heads and 0.05 dropout.

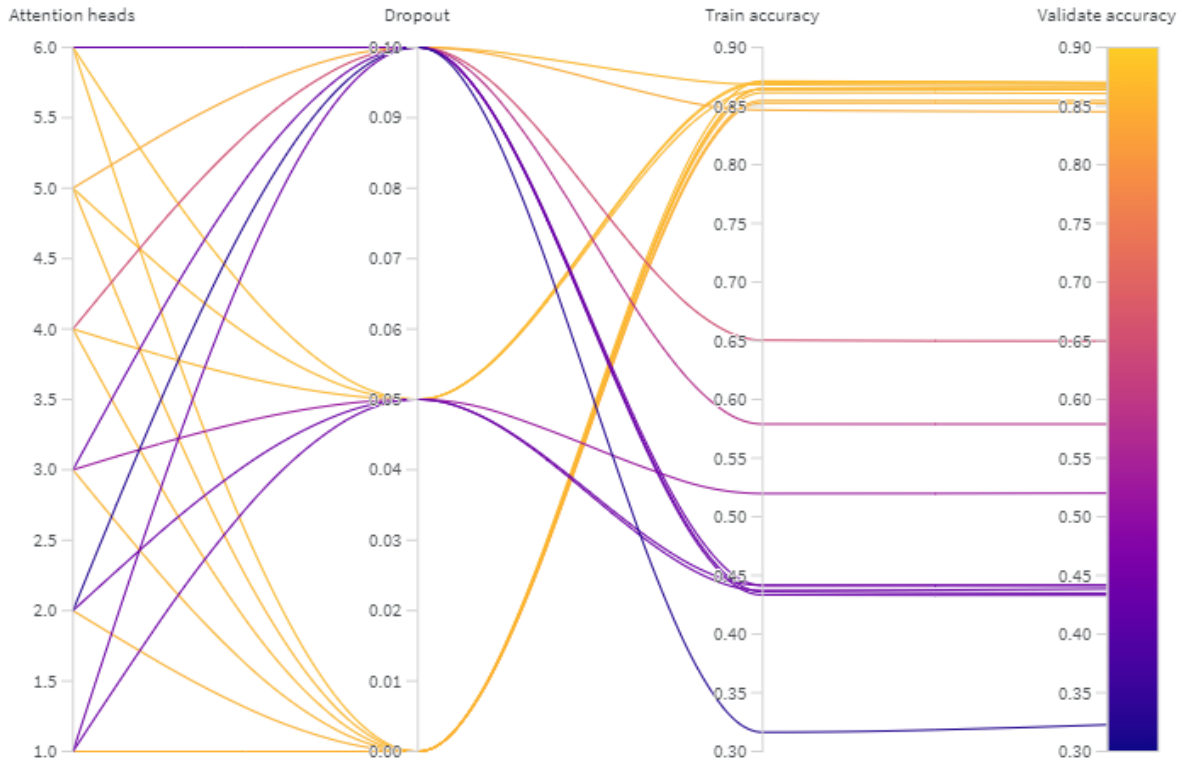

Fig. 43 Model with Visual transformer encoder grid search results

4.8.2. Best performing model confusion matrix and classification report

According to *Table 27 Model with Visual transformer encoder and our dataset accuracy report* and *Fig. 48 Model with Visual transformer encoder and our dataset confusion matrix* we see that similarly to pix2code results, this model predicts button colours deciding tokens the best compared to other models. However, it is the only model that achieves zero F1 scores for tokens. Oddly it also confuses various closing tags, and it predicts ‘</button>’, ‘’ instead of ‘</h4>’ and vice versa.

4.9. Experiments with our dataset and model with EfiecientNet encoder

This section will overview grid search results with our models and pix2code dataset and analyse best performing model accuracy.

4.9.1. Grid search results

According to *Fig. 44 Model with EfiecientNet encoder grid search results* highest validation accuracy was achieved by configuration of six attention heads and zero dropout. However, on average, the 0.05 dropout configuration reached higher validation accuracy than the zero dropout configurations.

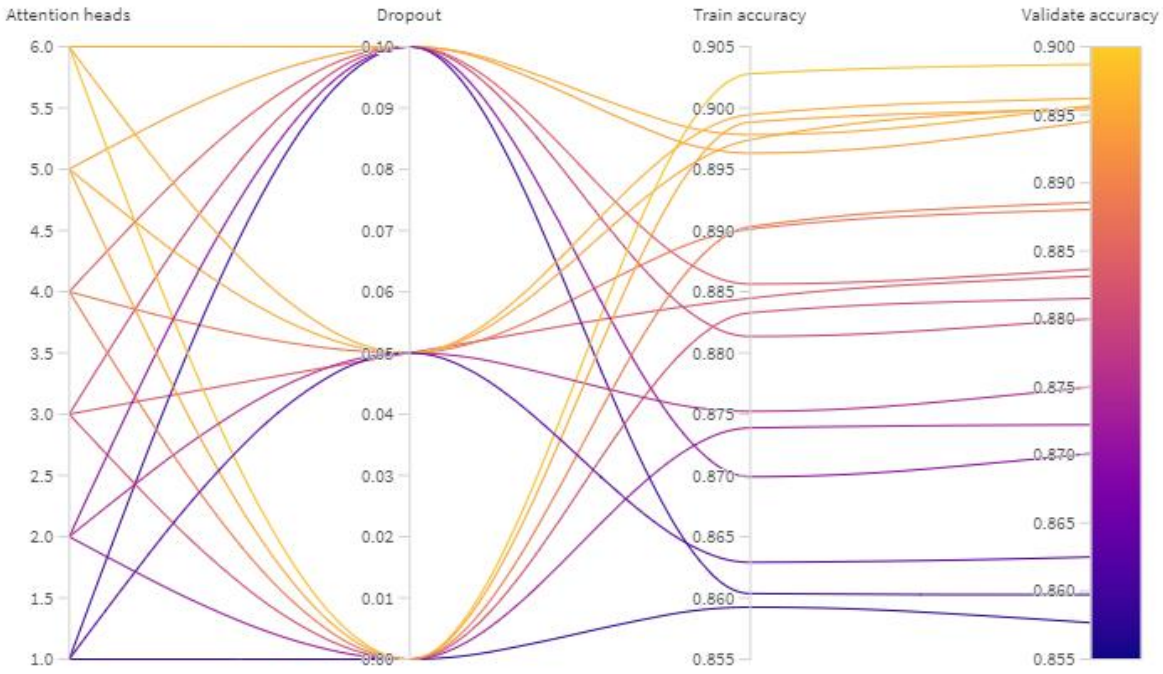

Fig. 44 Model with EfiecientNet encoder grid search results

4.9.2. Best performing model confusion matrix and classification report

According to *Table 28 Model with EfficientNet encoder and our dataset accuracy report* we can see and to *Fig. 49 Model with Efficientnet encoder and our dataset confusion matrix* we see that this model also struggles with predicting colours of buttons it has zero F1 scores for five tokens, which means that it never predicts them.

4.10. Experiments with our dataset and model with ResNet encoder

This section will overview grid search results with our models and pix2code dataset and analyse best performing model accuracy.

4.10.1. Grid search results

According to the *Fig. 45 Model with Resnet encoder grid search results, the* best performing model has 0 dropout and six attention heads. Similarly to EfiecientNet encoder results, configuration with 0.05 performed better than configurations without dropout.

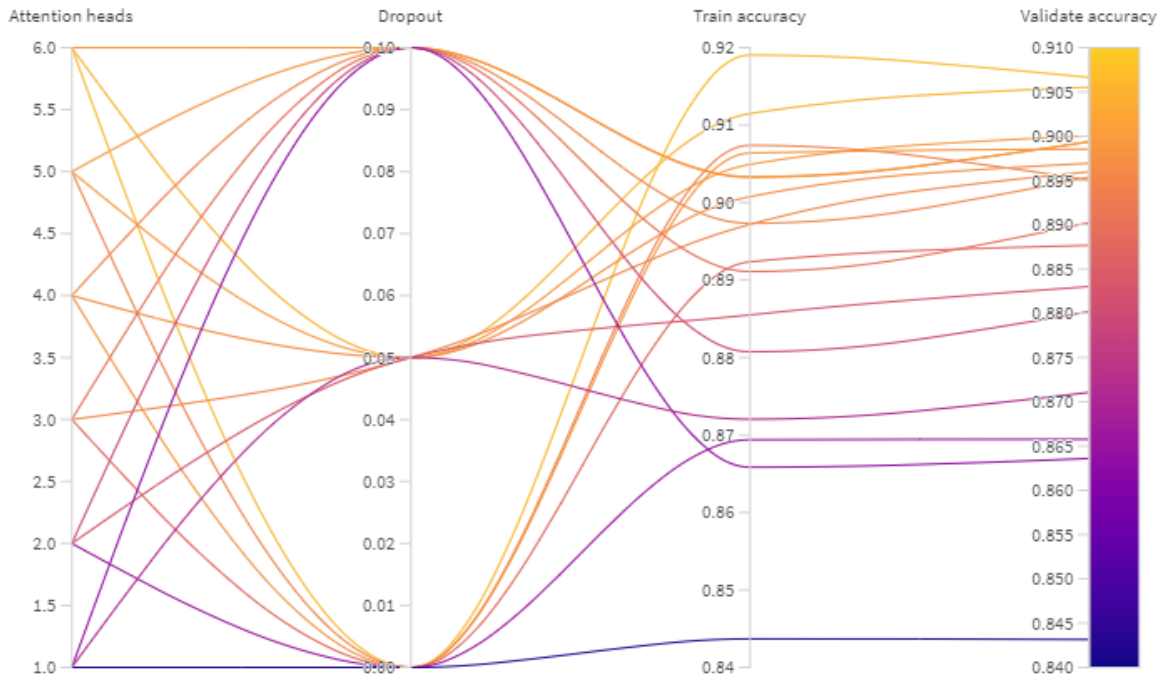

Fig. 45 Model with Resnet encoder grid search results

4.10.2. Best performing model confusion matrix and classification report

According to *Table 26 Model with Resnet encoder and our dataset accuracy report* and *Fig. 47 Model with Resnet encoder and our dataset confusion matrix* we see that it also mixes button colours predictions and receive zero F1 predictions for four tokens

4.11. Experiments with our dataset and

Like the pix2code dataset, the most accurate networks use VGG16 and Resnet Cnn architectures for their encoders. Also, having 0.05 on average allows networks to achieve higher or the same accuracy as having no dropout while being less overfit.

Table 16 Our dataset grid search top results

Network encoder	Number of heads	Dropout	Validate Accuracy	Test accuracy
VGG16	6	0.05	0.9233	0.9162
ResNet	6	0	0.919	0.9066
Vit16	6	0	0.8707	0.8701
EffNetB0	6	0	0.9028	0.8987

From *Table 29 Best performing modes different encoders and our dataset F1* accuracy comparison we can see that tokens which can be paired without alternatives (such as ‘justify-content:’ and ‘space-around;’ achieves higher F1 score, than those with alternatives (for example ‘background and linear-gradient(#fbb450,#f89406); or ‘linear-gradient(#62c462,#51a351);’ or ‘linear-gradient(#ee5f5b,#bd362f);’) and more possible tokens pair is the lower F1 score gets. Also, since this noticeable from both datasets, we can assume that model well predicts high-level features, such as the count and order of components. However, it struggles to predict low-level features, like component colour.

Conclusions

1. After analysing various image processing, language modelling, and image captioning networks. We came with two hypotheses: either Full transformer architecture for image captioning s or using image captioning with stacked attentions with cnn with pre-trained weights would yield the best result.
2. In this paper, we described an implementation software and rules used to generate the pix2html dataset. We generate a dataset with similar images but ~8.9 times longer sequences and a ~4.5 times broader vocabulary than the original pix2code dataset. This dataset will help to evaluate our solution performance on simple and more complicated datasets.
3.We successfully implemented and applied two different architectures for image captioning, which resulted in four different models. All were able to generate HTML from an image with accuracy higher than pix2code. We expected that the best-performing model would use a visual transformer in its image processing part. However, the results show that models with VGG16 decoder achieve the highest overall accuracy. This result could be explained via the VGG16 extracting higher-level features and not paying much attention to minor details, thus having a lower accuracy in image classification tasks but higher for our particular problem. We have also observed similarities and differences between various tokens predictions accuracies and noticed that the model that uses vision transformer predicts the most challenging part (green, orange and red buttons) better than cnn based encoder models. We have also noticed that most confused tokens are indicating colours for both our and pix2code datasets buttons. Furthermore, we noticed that predicting HTML sequence from images of size 224x224 pixels struggles to differentiate low-level features such as font size of 14px or 18px.
4.After comparing our solutions with already present ones, we found that our solution reaches higher accuracy with both our and pix2code datasets. We also noticed only a slight accuracy decrease (0.9355 with pix2code versus 0.9336 with our dataset).

List of references

1. Y. LeCun, et al. *Backpropagation Applied to Handwritten Zip Code Recognition.*

2. KRIZHEVSKY, A., SUTSKEVER, I. and HINTON, G. *ImageNet Classification with Deep Convolutional Neural Networks.* New York: ACM, May 24, 2017 Available from: http://dl.acm.org/citation.cfm?id=3065386 ABI/INFORM Collection China. ISBN 0001-0782. DOI 10.1145/3065386.

3. HE, K., ZHANG, X., REN, S. and SUN, J. *Deep Residual Learning for Image Recognition.* , Dec 10, 2015 Available from: https://www.openaire.eu/search/publication?articleId=od_____18::e7235b2295e7fd00c3555a8bfeb2c6b0.

4. TAN, M. and LE, Q.V. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks, May 28, 2019. Available from: https://arxiv.org/abs/1905.11946.

5. SZEGEDY, C., et al. *Going Deeper with Convolutions.* , Sep 16, 2014 Available from: https://www.openaire.eu/search/publication?articleId=od_____18::5df6d69237f08518896984361ca7485a.

6. SZEGEDY, C., et al. *Rethinking the Inception Architecture for Computer Vision.* Computer Vision and Pattern Recognition 2016, Dec 1, 2015 Available from: https://www.openaire.eu/search/publication?articleId=dedup_wf_001::e3ccf9935e81d95ffbc9afc5f36a2092.

7. Gao Huang, Zhuang Liu, VAN DER MAATEN, L. and WEINBERGER, K.Q. *Densely Connected Convolutional Networks.* IEEE, Jul 2017 Available from: https://ieeexplore.ieee.org/document/8099726 ISBN 1063-6919. DOI 10.1109/CVPR.2017.243.

8. CHOLLET, F. *Xception: Deep Learning with Depthwise Separable Convolutions.* IEEE, Jul 2017 Available from: https://ieeexplore.ieee.org/document/8099678 ISBN 1063-6919. DOI 10.1109/CVPR.2017.195.

9. YU, F., KOLTUN, V. and FUNKHOUSER, T. *Dilated Residual Networks.* IEEE, Jul 2017 Available from: https://ieeexplore.ieee.org/document/8099558 ISBN 1063-6919. DOI 10.1109/CVPR.2017.75.

10. CAI, H., ZHU, L. and HAN, S. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware, Dec 2, 2018. Available from: https://arxiv.org/abs/1812.00332.

11. ZOPH, B., VASUDEVAN, V., SHLENS, J. and LE, Q.V. *Learning Transferable Architectures for Scalable Image Recognition.* IEEE, Jun 2018 Available from: https://ieeexplore.ieee.org/document/8579005 DOI 10.1109/CVPR.2018.00907.

12. SZEGEDY, C., IOFFE, S., VANHOUCKE, V. and ALEMI, A. *Inception-V4, Inception-ResNet and the Impact of Residual Connections on Learning.* , Feb 23, 2016 Available from: https://www.openaire.eu/search/publication?articleId=od_____18::44ac8f8822c35366d643221ef4b97e42.

13. WU, B., et al. Visual Transformers: Token-Based Image Representation and Processing for Computer Vision, Jun 05, 2020. Available from: https://arxiv.org/abs/2006.03677.

14. CHO, K., et al. *Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation.* Association for Computational Linguistics (ACL), 2014 Available from: https://search.datacite.org/works/10.3115/v1/d14-1179 DOI 10.3115/v1/d14-1179.

15. HOCHREITER, S. and SCHMIDHUBER, J. Long Short-Term Memory. *Neural Computation*, Nov, 1997, vol. 9, no. 8. pp. 1735-1780. Available from: https://search.datacite.org/works/10.1162/neco.1997.9.8.1735 MEDLINE. ISSN 1530-888X. DOI 10.1162/neco.1997.9.8.1735.

16. CHUNG, J., GULCEHRE, C., CHO, K. and BENGIO, Y. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, Dec 11, 2014. Available from: https://arxiv.org/abs/1412.3555.

17. GOOGLE, I.S., ORIOL, V., Google and LE GOOGLE, Q.V. *Sequence to Sequence Learning with Neural Networks.* , -12-14, 2014.

18. BAHDANAU, D., CHO, K. and BENGIO, Y. *Published as a Conference Paper at ICLR 2015 NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE*.

19. VASWANI, A., et al. Attention is all You Need, Jun 12, 2017. Available from: https://arxiv.org/abs/1706.03762.

20. LEE, J., MANSIMOV, E. and CHO, K. Deterministic Non-Autoregressive Neural Sequence Modeling by Iterative Refinement, Feb 19, 2018. Available from: https://arxiv.org/abs/1802.06901.

21. KITAEV, N., KAISER, Ł and LEVSKAYA, A. Reformer: The Efficient Transformer, Jan 13, 2020. Available from: https://arxiv.org/abs/2001.04451.

22. AKBARI, H., et al. Multi-Level Multimodal Common Semantic Space for Image-Phrase Grounding, Nov 28, 2018. Available from: https://arxiv.org/abs/1811.11683.

23. HOSSAIN, M.Z., SOHEL, F., SHIRATUDDIN, M.F. and LAGA, H. A Comprehensive Survey of Deep Learning for Image Captioning, Oct 6, 2018. Available from: https://arxiv.org/abs/1810.04020.

24. XU, K., et al. *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention.* , Feb 10, 2015 Available from: https://www.openaire.eu/search/publication?articleId=od_____18::938e5c5c3b2dd7812b5d23e3719d7b86.

25. ZHU, X., et al. Captioning Transformer with Stacked Attention Modules. *Applied Sciences*, May 07, 2018, vol. 8, no. 5. pp. 739. Available from: https://search.proquest.com/docview/2321853478 CrossRef. ISSN 2076-3417. DOI 10.3390/app8050739.

26. HERDADE, S., KAPPELER, A., BOAKYE, K. and SOARES, J. Image Captioning: Transforming Objects into Words, Jun 13, 2019. Available from: https://arxiv.org/abs/1906.05963.

27. CORNIA, M., STEFANINI, M., BARALDI, L. and CUCCHIARA, R. *Meshed-Memory Transformer for Image Captioning.* IEEE, Jun 2020 Available from: https://ieeexplore.ieee.org/document/9157222 DOI 10.1109/CVPR42600.2020.01059.

28. LIU, W., et al. CPTR: Full Transformer Network for Image Captioning, Jan 26, 2021. Available from: https://arxiv.org/abs/2101.10804.

29. ZHU, Z., XUE, Z. and YUAN, Z. Automatic Graphics Program Generation using Attention-Based Hierarchical Decoder, Oct 26, 2018. Available from: https://arxiv.org/abs/1810.11536.

30. LOSHCHILOV, I. and HUTTER, F. Decoupled Weight Decay Regularization, Nov 14, 2017. Available from: https://arxiv.org/abs/1711.05101.

31. SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A. and SALAKHUTDINOV, R. *Machine Learning; Reports Summarize Machine Learning Study Results from University of Toronto (Dropout: A Simple Way to Prevent Neural Networks from Overfitting).* Atlanta: NewsRx, Dec 15, 2014 Available from: https://search.proquest.com/docview/1634927573 ISBN 1944-1851.

Appendices

Appendix 1. Grid search results with pix2code dataset

Table 17 Model with VGG16 encoder and pix2code dataset grid search results

Dropout	Heads	Train accuracy	Validate accuracy
0.05	6	0.942761	0.940367
0	6	0.941713	0.939184
0.05	5	0.940158	0.93779
0.05	4	0.93812	0.935371
0.1	6	0.937464	0.934996
0.1	4	0.936266	0.934479
0	5	0.933551	0.931491
0	4	0.931899	0.930187
0.1	5	0.930769	0.928352
0.05	3	0.930001	0.928277
0	3	0.92608	0.924464
0.1	3	0.926379	0.923805
0.05	2	0.917131	0.91585
0	2	0.916152	0.914772
0.1	2	0.91277	0.911318
0.05	1	0.89027	0.889191
0.1	1	0.881294	0.880794
0	1	0.879801	0.877595

Table 18 Model with Resnet and pix2code grid search results

Dropout	Heads	Train accuracy	Validate accuracy
0.05	6	0.939252	0.930322
0	6	0.942379	0.928951

0	5	0.93606	0.922472
0.05	5	0.935074	0.922225
0	4	0.934111	0.921318
0.1	5	0.925185	0.917498
0.1	6	0.927398	0.917318
0.05	4	0.929919	0.915378
0.05	3	0.914808	0.906217
0.1	4	0.913213	0.905918
0	2	0.912211	0.902075
0	3	0.919809	0.900397
0.1	3	0.906805	0.900097
0.05	2	0.903878	0.894876
0.1	2	0.895723	0.888112
0.05	1	0.892286	0.886637
0.1	1	0.878405	0.873213
0	1	0.877206	0.869019

Table 19 Model with Visual transformer encoder and pix2code dataset grid search results

Dropout	**Heads**	**Train accuracy**	**Validate accuracy**
0	6	0.864255	0.863326
0	5	0.862172	0.861011
0	4	0.858793	0.856629
0.05	6	0.855646	0.854592
0	3	0.854709	0.853266
0.05	5	0.852991	0.852165
0.1	6	0.851774	0.851723

0.1	5	0.85047	0.850157
0	2	0.85121	0.850022
0	1	0.844478	0.844487
0.05	4	0.834355	0.834921
0.1	4	0.654628	0.655828
0.05	3	0.603535	0.601993
0.05	1	0.439176	0.436996
0.1	3	0.389298	0.386082
0.1	1	0.380948	0.381678
0.05	2	0.243837	0.244936
0.1	2	0.188221	0.187131

Table 20 Model with Eficientnet encoder and pix2code dataset grid search results

Dropout	**Heads**	**Train accuracy**	**Validate accuracy**
0	6	0.88462	0.878801
0	5	0.882465	0.876487
0	4	0.880297	0.875311
0.05	6	0.877305	0.873348
0.05	5	0.876633	0.872742
0	3	0.872947	0.868884
0.05	4	0.868082	0.866285
0.1	6	0.868876	0.866232
0.1	4	0.86489	0.862442
0.05	3	0.864305	0.861715
0	2	0.865079	0.861655
0.1	5	0.863932	0.861026

0.05	2	0.859307	0.857363
0.1	3	0.856623	0.855835
0.1	2	0.855525	0.8537
0.05	1	0.852254	0.851453
0	1	0.853836	0.851176
0.1	1	0.848349	0.847206

Appendix 2. Grid search results with our dataset

Table 21 Model with EfiecientNet encoder and our dataset grid search results

Dropout	**Heads**	**Train accuracy**	**Validate_accuracy**
0	6	0.902788	0.898702
0.05	6	0.899456	0.896196
0.1	6	0.897817	0.895667
0.05	5	0.897399	0.89556
0	5	0.898839	0.895377
0.1	5	0.896316	0.894511
0	4	0.890299	0.888553
0.05	4	0.890119	0.888036
0.1	4	0.885628	0.883659
0.05	3	0.884464	0.883135
0	3	0.88328	0.881503
0.1	3	0.881352	0.880012
0.05	2	0.875236	0.875008
0	2	0.873882	0.872221
0.1	2	0.869894	0.870111
0.05	1	0.862907	0.862508

0.1	1	0.860355	0.859741
0	1	0.859249	0.857682

Table 22 Model with VGG16 encoder and our dataset grid search results

Dropout	**Heads**	**Train accuracy**	**Validate accuracy**
0.05	5	0.923273	0.916224
0	6	0.925481	0.915613
0.05	6	0.924408	0.915107
0	5	0.921609	0.914064
0.1	6	0.920465	0.912556
0.05	4	0.916282	0.911488
0.1	5	0.913407	0.906987
0	4	0.912149	0.906539
0.1	4	0.908158	0.904542
0.05	3	0.907003	0.902743
0.1	3	0.901336	0.898486
0	3	0.903837	0.897839
0.05	2	0.898988	0.896409
0.1	2	0.893162	0.891781
0	2	0.885546	0.881367
0.05	1	0.871992	0.87169
0.1	1	0.866007	0.864246
0	1	0.800499	0.800454

Table 23 Model with Resnet encoder and our dataset grid search results

Dropout	**Heads**	**Train accuracy**	**Validate accuracy**
0	6	0.919045	0.906634

0.05	6	0.911493	0.905534
0.05	5	0.904925	0.900028
0.1	6	0.903257	0.899408
0.1	5	0.903351	0.899365
0	4	0.906398	0.898538
0.05	4	0.900766	0.896932
0.05	3	0.897189	0.895973
0.1	4	0.89733	0.89538
0	5	0.907392	0.895048
0.1	3	0.891107	0.890251
0	3	0.892362	0.887668
0.05	2	0.885501	0.883014
0.1	2	0.880744	0.880193
0.05	1	0.872037	0.871063
0	2	0.869387	0.865764
0.1	1	0.865844	0.863594
0	1	0.843682	0.843148

Table 24 Model with vision transformer encoder and our dataset grid search results

Dropout	**Heads**	**Train accuracy**	**Validate accuracy**
0.05	6	0.870698	0.870102
0	6	0.871257	0.869333
0.05	5	0.86944	0.868458
0.05	4	0.864151	0.864754
0	4	0.864459	0.86385
0	5	0.864874	0.863842

0	3	0.861395	0.860714
0	2	0.854681	0.854519
0	1	0.852632	0.852334
0.1	5	0.846624	0.845166
0.1	4	0.650462	0.649752
0.05	3	0.519767	0.520078
0.1	1	0.44156	0.44158
0.05	1	0.44156	0.44158
0.05	2	0.437145	0.438595
0.1	3	0.435996	0.434909
0.1	2	0.316371	0.322737

Appendix 3. Most accurate models with our dataset confusion matrixes and accuracy reports

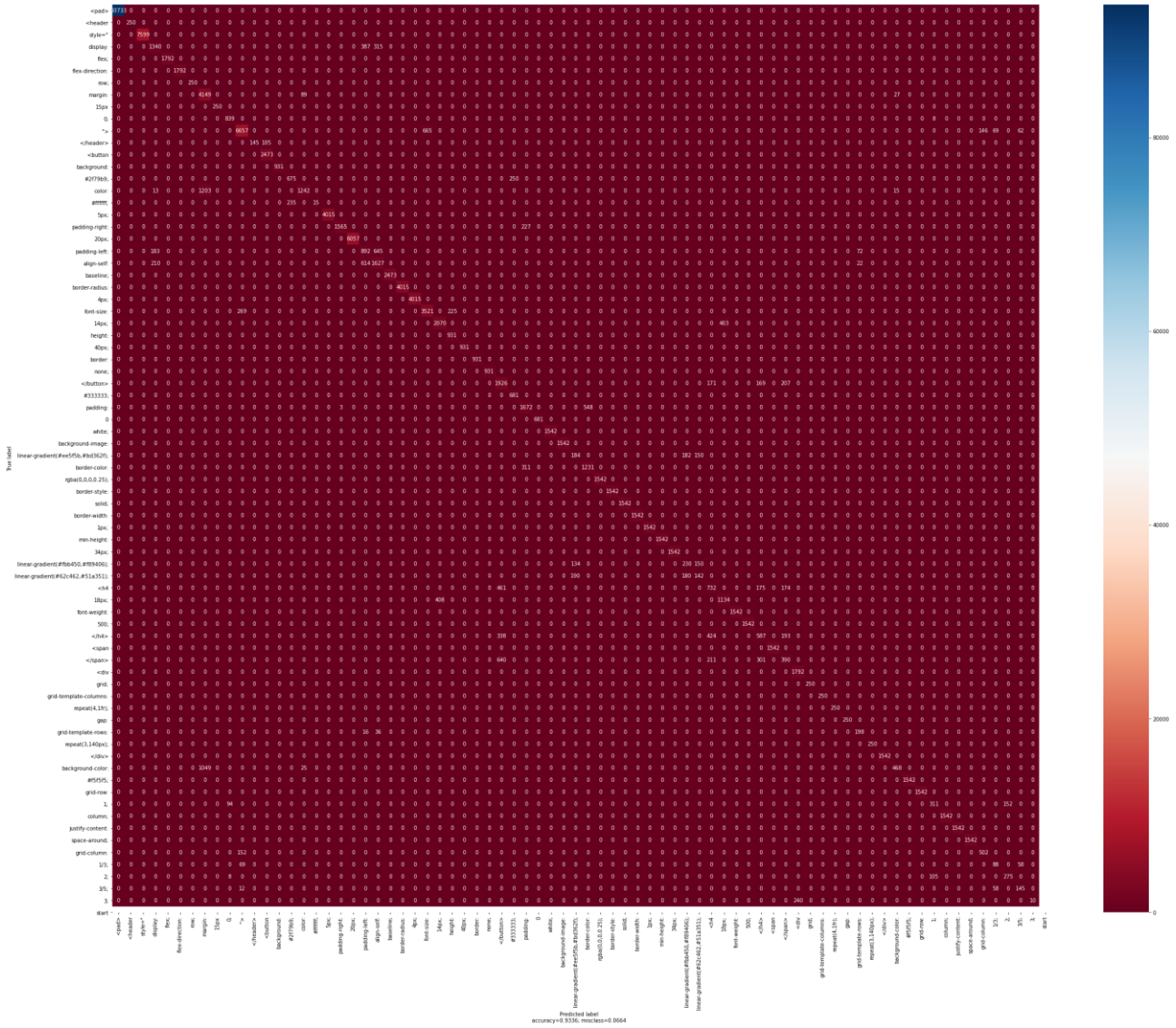

Fig. 46 Model with VGG16 encoder and our dataset confusion matrix

Table 25 Model with VGG16 encoder and our dataset precision report

Token	Precision	Recall	F1	Support
0	1.0000	1.0000	1.0000	93733
1	1.0000	1.0000	1.0000	250
2	1.0000	1.0000	1.0000	7599
3	0.7675	0.6562	0.7075	2042
4	1.0000	1.0000	1.0000	1792
5	1.0000	1.0000	1.0000	1792
6	1.0000	1.0000	1.0000	250
7	0.6482	0.9728	0.7780	4265
8	1.0000	1.0000	1.0000	250
9	0.8916	0.9906	0.9385	847
10	0.9299	0.8760	0.9022	7599
11	1.0000	0.5800	0.7342	250

12	0.9593	1.0000	0.9792	2473
13	1.0000	1.0000	1.0000	931
14	0.7418	0.7250	0.7333	931
15	0.9159	0.5022	0.6487	2473
16	0.7143	0.0600	0.1107	250
17	1.0000	1.0000	1.0000	4015
18	0.9981	0.8733	0.9315	1792
19	1.0000	1.0000	1.0000	6057
20	0.4673	0.4978	0.4820	1792
21	0.6203	0.6579	0.6385	2473
22	1.0000	1.0000	1.0000	2473
23	1.0000	1.0000	1.0000	4015
24	1.0000	1.0000	1.0000	4015
25	0.8411	0.8770	0.8587	4015
26	0.8354	0.8370	0.8362	2473
27	0.8054	1.0000	0.8922	931
28	1.0000	1.0000	1.0000	931
29	1.0000	1.0000	1.0000	931
30	1.0000	1.0000	1.0000	931
31	0.5724	0.7788	0.6598	2473
32	0.7315	1.0000	0.8449	681
33	0.7566	0.7521	0.7543	2223
34	1.0000	1.0000	1.0000	681
35	1.0000	1.0000	1.0000	1542
36	1.0000	1.0000	1.0000	1542
37	0.3622	0.3566	0.3594	516
38	0.6920	0.7983	0.7413	1542
39	1.0000	1.0000	1.0000	1542
40	1.0000	1.0000	1.0000	1542
41	1.0000	1.0000	1.0000	1542
42	1.0000	1.0000	1.0000	1542
43	1.0000	1.0000	1.0000	1542
44	1.0000	1.0000	1.0000	1542
45	1.0000	1.0000	1.0000	1542
46	0.3885	0.4475	0.4159	514
47	0.3213	0.2773	0.2977	512

48	0.4759	0.4747	0.4753	1542
49	0.7378	0.7354	0.7366	1542
50	1.0000	1.0000	1.0000	1542
51	1.0000	1.0000	1.0000	1542
52	0.4765	0.3807	0.4232	1542
53	1.0000	1.0000	1.0000	1542
54	0.4046	0.2529	0.3113	1542
55	0.8819	1.0000	0.9372	1792
56	1.0000	1.0000	1.0000	250
57	1.0000	1.0000	1.0000	250
58	1.0000	1.0000	1.0000	250
59	1.0000	1.0000	1.0000	250
60	0.6781	0.7920	0.7306	250
61	1.0000	1.0000	1.0000	250
62	1.0000	1.0000	1.0000	1542
63	0.9176	0.3035	0.4561	1542
64	1.0000	1.0000	1.0000	1542
65	1.0000	1.0000	1.0000	1542
66	0.7335	0.5583	0.6340	557
67	1.0000	1.0000	1.0000	1542
68	1.0000	1.0000	1.0000	1542
69	1.0000	1.0000	1.0000	1542
70	0.7747	0.7676	0.7711	654
71	0.4093	0.4093	0.4093	215
72	0.6440	0.7088	0.6748	388
73	0.5472	0.6744	0.6042	215

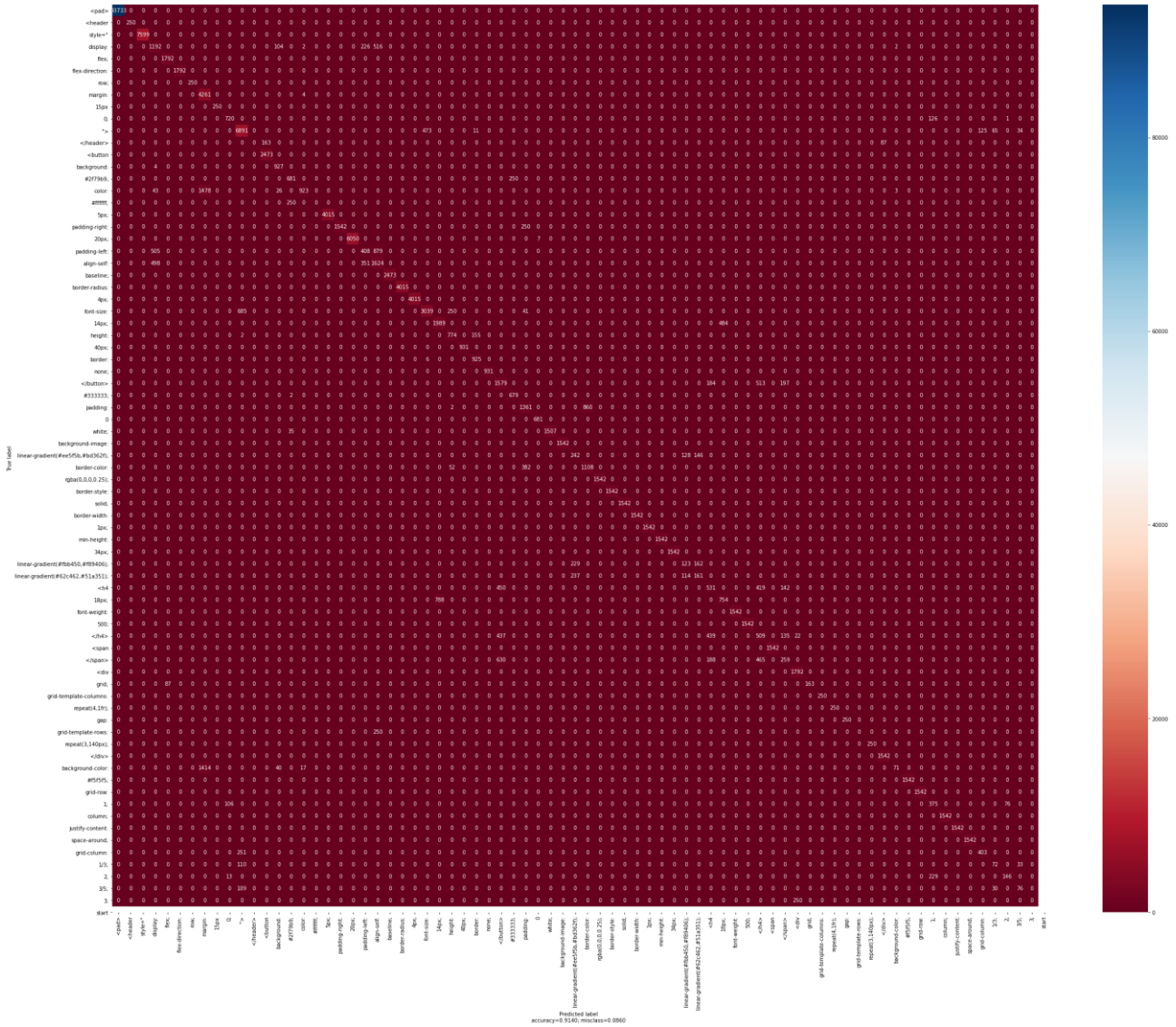

Fig. 47 Model with Resnet encoder and our dataset confusion matrix

Table 26 Model with Resnet encoder and our dataset accuracy report

Token	Precision	Recall	F1	Support
0	1.0000	1.0000	1.0000	93733
1	1.0000	1.0000	1.0000	250
2	1.0000	1.0000	1.0000	7599
3	0.5317	0.5837	0.5565	2042
4	0.9537	1.0000	0.9763	1792
5	1.0000	1.0000	1.0000	1792
6	1.0000	1.0000	1.0000	250
7	0.5957	0.9991	0.7464	4265
8	1.0000	1.0000	1.0000	250
9	0.8582	0.8501	0.8541	847
10	0.8562	0.9068	0.8808	7599
11	0.0000	0.0000	0.0000	250

12	0.9382	1.0000	0.9681	2473
13	0.8450	0.9957	0.9142	931
14	0.7035	0.7315	0.7172	931
15	0.9757	0.3732	0.5399	2473
16	0.0000	0.0000	0.0000	250
17	1.0000	1.0000	1.0000	4015
18	1.0000	0.8605	0.9250	1792
19	1.0000	0.9988	0.9994	6057
20	0.4142	0.2277	0.2938	1792
21	0.4968	0.6567	0.5657	2473
22	1.0000	1.0000	1.0000	2473
23	1.0000	1.0000	1.0000	4015
24	1.0000	1.0000	1.0000	4015
25	0.8638	0.7569	0.8068	4015
26	0.7162	0.8043	0.7577	2473
27	0.7180	0.8314	0.7705	931
28	1.0000	1.0000	1.0000	931
29	0.8478	0.9936	0.9149	931
30	1.0000	1.0000	1.0000	931
31	0.5100	0.6385	0.5671	2473
32	0.7309	0.9971	0.8435	681
33	0.6691	0.6122	0.6394	2223
34	0.9898	1.0000	0.9949	681
35	1.0000	0.9773	0.9885	1542
36	1.0000	1.0000	1.0000	1542
37	0.3418	0.4690	0.3954	516
38	0.5630	0.7185	0.6313	1542
39	1.0000	1.0000	1.0000	1542
40	1.0000	1.0000	1.0000	1542
41	1.0000	1.0000	1.0000	1542
42	1.0000	1.0000	1.0000	1542
43	1.0000	1.0000	1.0000	1542
44	1.0000	1.0000	1.0000	1542
45	1.0000	1.0000	1.0000	1542
46	0.3370	0.2393	0.2799	514
47	0.3433	0.3145	0.3282	512

48	0.3957	0.3444	0.3682	1542
49	0.6090	0.4890	0.5424	1542
50	1.0000	1.0000	1.0000	1542
51	1.0000	1.0000	1.0000	1542
52	0.2671	0.3301	0.2952	1542
53	1.0000	1.0000	1.0000	1542
54	0.3533	0.1680	0.2277	1542
55	0.8682	1.0000	0.9295	1792
56	1.0000	0.6520	0.7893	250
57	1.0000	1.0000	1.0000	250
58	1.0000	1.0000	1.0000	250
59	1.0000	1.0000	1.0000	250
60	0.0000	0.0000	0.0000	250
61	1.0000	1.0000	1.0000	250
62	0.9466	1.0000	0.9726	1542
63	0.9342	0.0460	0.0878	1542
64	1.0000	1.0000	1.0000	1542
65	1.0000	1.0000	1.0000	1542
66	0.5137	0.6732	0.5828	557
67	1.0000	1.0000	1.0000	1542
68	1.0000	1.0000	1.0000	1542
69	1.0000	1.0000	1.0000	1542
70	0.7633	0.6162	0.6819	654
71	0.4311	0.3349	0.3770	215
72	0.6547	0.3763	0.4779	388
73	0.5315	0.3535	0.4246	215

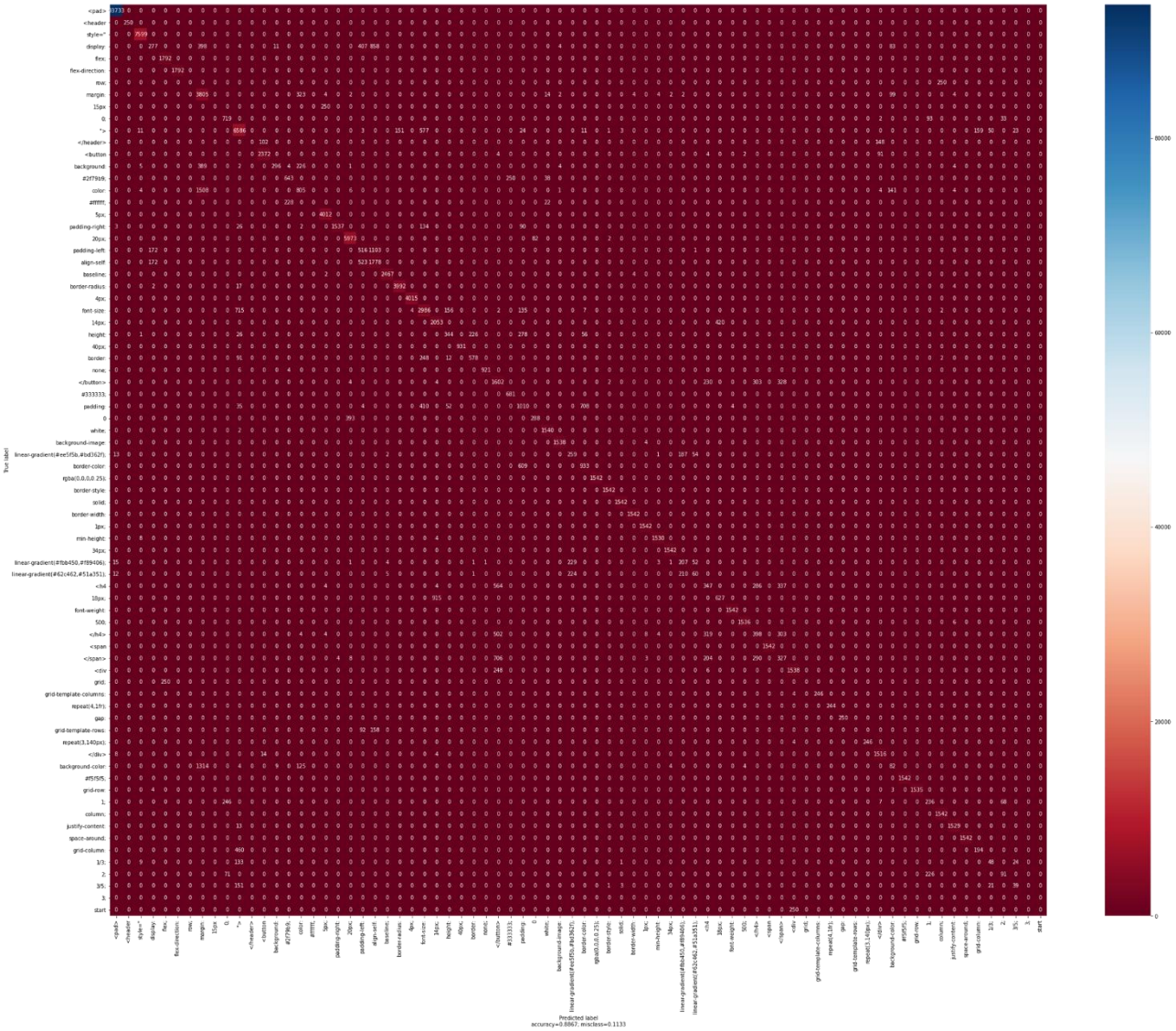

Fig. 48 Model with Visual transformer encoder and our dataset confusion matrix

Table 27 Model with Visual transformer encoder and our dataset accuracy report

Token	Precision	Recall	F1	Support
0	0.9982	1.0000	0.9991	93733
1	0.9690	1.0000	0.9843	250
2	0.9927	0.9991	0.9959	7599
3	0.5027	0.2321	0.3176	2042
4	0.8771	0.9961	0.9328	1792
5	1.0000	1.0000	1.0000	1792
6	1.0000	0.9760	0.9879	250
7	0.5164	0.8996	0.6562	4265
8	0.0000	0.0000	0.0000	250
9	0.6298	0.8194	0.7122	847
10	0.7379	0.8741	0.8002	7599
11	0.0000	0.0000	0.0000	250

12	0.9318	0.8900	0.9104	2473
13	0.8389	0.6208	0.7136	931
14	0.6897	0.6874	0.6885	931
15	0.6246	0.3182	0.4216	2473
16	0.0000	0.0000	0.0000	250
17	0.9307	0.9963	0.9623	4015
18	0.9961	0.8560	0.9208	1792
19	0.9247	0.9982	0.9601	6057
20	0.3868	0.2935	0.3338	1792
21	0.4627	0.7315	0.5668	2473
22	1.0000	0.9964	0.9982	2473
23	0.9825	0.9940	0.9882	4015
24	1.0000	0.9973	0.9986	4015
25	0.6763	0.7452	0.7091	4015
26	0.6394	0.7214	0.6779	2473
27	0.6082	0.4286	0.5028	931
28	1.0000	1.0000	1.0000	931
29	0.6747	0.3899	0.4942	931
30	1.0000	1.0000	1.0000	931
31	0.3942	0.6268	0.4840	2473
32	0.7304	0.9868	0.8395	681
33	0.4387	0.3527	0.3910	2223
34	0.9478	0.3730	0.5353	681
35	0.9662	0.9442	0.9551	1542
36	1.0000	0.9981	0.9990	1542
37	0.3869	0.3081	0.3430	516
38	0.5665	0.6543	0.6073	1542
39	1.0000	0.9916	0.9958	1542
40	1.0000	0.9929	0.9964	1542
41	1.0000	0.9916	0.9958	1542
42	0.9916	0.9968	0.9942	1542
43	0.9948	0.9857	0.9902	1542
44	1.0000	0.9916	0.9958	1542
45	0.9878	0.9968	0.9923	1542
46	0.3577	0.3716	0.3645	514
47	0.3707	0.4004	0.3850	512

48	0.2985	0.2853	0.2918	1542
49	0.4376	0.3457	0.3862	1542
50	0.9929	0.9981	0.9955	1542
51	1.0000	0.9916	0.9958	1542
52	0.3564	0.1770	0.2366	1542
53	1.0000	0.9909	0.9954	1542
54	0.2169	0.1913	0.2033	1542
55	0.8567	0.8571	0.8569	1792
56	0.0000	0.0000	0.0000	250
57	1.0000	1.0000	1.0000	250
58	1.0000	1.0000	1.0000	250
59	1.0000	1.0000	1.0000	250
60	0.0000	0.0000	0.0000	250
61	1.0000	1.0000	1.0000	250
62	0.9270	0.9968	0.9606	1542
63	0.2103	0.0292	0.0513	1542
64	1.0000	0.9838	0.9918	1542
65	1.0000	0.9981	0.9990	1542
66	0.3654	0.2998	0.3294	557
67	0.9974	0.9955	0.9964	1542
68	1.0000	0.9981	0.9990	1542
69	1.0000	1.0000	1.0000	1542
70	0.6131	0.1865	0.2860	654
71	0.4500	0.0419	0.0766	215
72	0.5027	0.2397	0.3246	388
73	0.2632	0.1163	0.1613	215
74	0.000	0.000	0.000	250

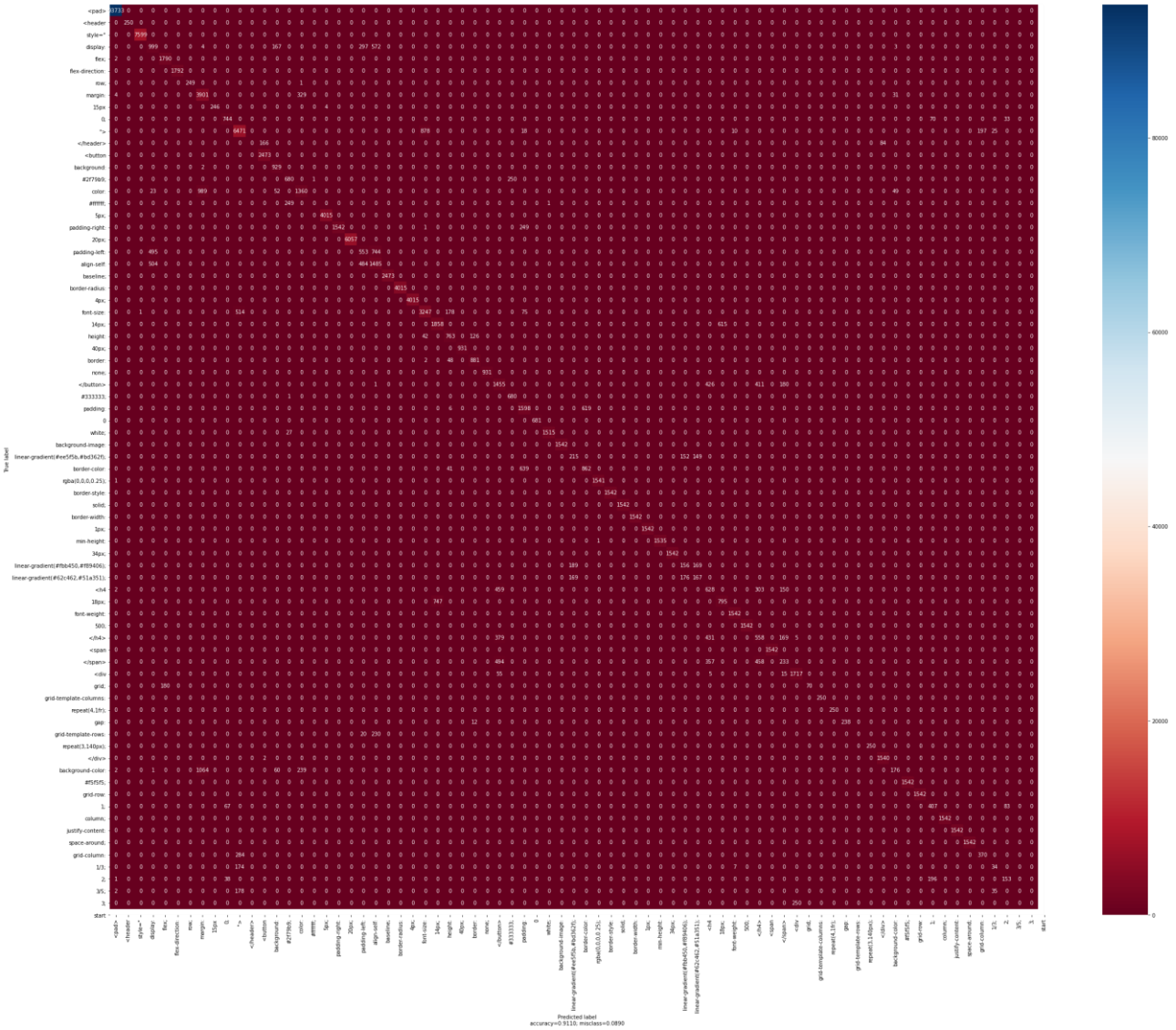

Fig. 49 Model with Efficientnet encoder and our dataset confusion matrix

Table 28 Model with EfficientNet encoder and our dataset accuracy report

token	precision	recall	f1	support
0	0.9999	1.0000	0.9999	93733
1	1.0000	1.0000	1.0000	250
2	0.9999	1.0000	0.9999	7599
3	0.4941	0.4892	0.4916	2042
4	0.9086	0.9989	0.9516	1792
5	1.0000	1.0000	1.0000	1792
6	1.0000	0.9960	0.9980	250
7	0.6545	0.9147	0.7630	4265

8	1.0000	0.9840	0.9919	250
9	0.8763	0.8784	0.8774	847
10	0.8491	0.8516	0.8503	7599
11	0.0000	0.0000	0.0000	250
12	0.9364	1.0000	0.9671	2473
13	0.7690	0.9979	0.8686	931
14	0.7106	0.7304	0.7203	931
15	0.7050	0.5499	0.6179	2473
16	0.0000	0.0000	0.0000	250
17	0.9990	1.0000	0.9995	4015
18	1.0000	0.8605	0.9250	1792
19	1.0000	1.0000	1.0000	6057
20	0.4084	0.3086	0.3516	1792
21	0.4898	0.6005	0.5395	2473
22	1.0000	1.0000	1.0000	2473
23	1.0000	1.0000	1.0000	4015
24	1.0000	1.0000	1.0000	4015
25	0.7787	0.8087	0.7934	4015
26	0.7132	0.7513	0.7318	2473
27	0.7365	0.8195	0.7758	931
28	1.0000	1.0000	1.0000	931
29	0.8646	0.9463	0.9036	931
30	1.0000	1.0000	1.0000	931
31	0.5120	0.5884	0.5475	2473
32	0.7312	0.9985	0.8442	681

33	0.6196	0.7188	0.6656	2223
34	1.0000	1.0000	1.0000	681
35	0.9993	0.9825	0.9908	1542
36	1.0000	1.0000	1.0000	1542
37	0.3752	0.4167	0.3949	516
38	0.5820	0.5590	0.5703	1542
39	0.9994	0.9994	0.9994	1542
40	1.0000	1.0000	1.0000	1542
41	1.0000	1.0000	1.0000	1542
42	1.0000	1.0000	1.0000	1542
43	1.0000	1.0000	1.0000	1542
44	1.0000	0.9955	0.9977	1542
45	1.0000	1.0000	1.0000	1542
46	0.3223	0.3035	0.3126	514
47	0.3443	0.3262	0.3350	512
48	0.3400	0.4073	0.3706	1542
49	0.5638	0.5156	0.5386	1542
50	0.9891	1.0000	0.9945	1542
51	1.0000	1.0000	1.0000	1542
52	0.3225	0.3619	0.3411	1542
53	1.0000	1.0000	1.0000	1542
54	0.3119	0.1511	0.2036	1542
55	0.8707	0.9581	0.9123	1792
56	1.0000	0.2800	0.4375	250
57	1.0000	1.0000	1.0000	250

58	1.0000	1.0000	1.0000	250
59	1.0000	0.9520	0.9754	250
60	0.0000	0.0000	0.0000	250
61	1.0000	1.0000	1.0000	250
62	0.9483	0.9987	0.9728	1542
63	0.6795	0.1141	0.1954	1542
64	0.9961	1.0000	0.9981	1542
65	1.0000	1.0000	1.0000	1542
66	0.6048	0.7307	0.6618	557
67	1.0000	1.0000	1.0000	1542
68	1.0000	1.0000	1.0000	1542
69	1.0000	1.0000	1.0000	1542
70	0.6526	0.5657	0.6061	654
71	0.3617	0.1581	0.2201	215
72	0.5688	0.3943	0.4658	388
73	0.000	0.0000	0.0000	215
74	0.000	0.000	0.000	250

Table 29 Best performing modes different encoders and our dataset F1 accuracy comparison

Token	**VGG16**	**Resnet**	**Efiecient**	**VIT**
<pad>	1.0000	1	0.9999	0.9991
<header	1.0000	1	1	0.9843
style="	1.0000	1	0.9999	0.9959
display:	0.7075	0.5565	0.4916	0.3176
flex;	1.0000	0.9763	0.9516	0.9328
flex-direction:	1.0000	1	1	1
row;	1.0000	1	0.998	0.9879
margin:	0.7780	0.7464	0.763	0.6562

15px	1.0000	1	0.9919	0
0;	0.9385	0.8541	0.8774	0.7122
">	0.9022	0.8808	0.8503	0.8002
</header>	0.7342	0	0	0
<button	0.9792	0.9681	0.9671	0.9104
background:	1.0000	0.9142	0.8686	0.7136
#2f79b9;	0.7333	0.7172	0.7203	0.6885
color:	0.6487	0.5399	0.6179	0.4216
#ffffff;	0.1107	0	0	0
5px;	1.0000	1	0.9995	0.9623
padding-right:	0.9315	0.925	0.925	0.9208
20px;	1.0000	0.9994	1	0.9601
padding-left:	0.4820	0.2938	0.3516	0.3338
align-self:	0.6385	0.5657	0.5395	0.5668
baseline;	1.0000	1	1	0.9982
border-radius:	1.0000	1	1	0.9882
4px;	1.0000	1	1	0.9986
font-size:	0.8587	0.8068	0.7934	0.7091
14px;	0.8362	0.7577	0.7318	0.6779
height:	0.8922	0.7705	0.7758	0.5028
40px;	1.0000	1	1	1
border:	1.0000	0.9149	0.9036	0.4942
none;	1.0000	1	1	1
</button>	0.6598	0.5671	0.5475	0.484
#333333;	0.8449	0.8435	0.8442	0.8395
padding:	0.7543	0.6394	0.6656	0.391
0	1.0000	0.9949	1	0.5353
white;	1.0000	0.9885	0.9908	0.9551
background-image:	1.0000	1	1	0.999
linear-gradient(#ee5f5b,#bd362f);	0.3594	0.3954	0.3949	0.343
border-color:	0.7413	0.6313	0.5703	0.6073

rgba(0,0,0,0.25);	1.0000	1	0.9994	0.9958
border-style:	1.0000	1	1	0.9964
solid;	1.0000	1	1	0.9958
border-width:	1.0000	1	1	0.9942
1px;	1.0000	1	1	0.9902
min-height:	1.0000	1	0.9977	0.9958
34px;	1.0000	1	1	0.9923
linear-gradient(#fbb450,#f89406);	0.4159	0.2799	0.3126	0.3645
linear-gradient(#62c462,#51a351);	0.2977	0.3282	0.335	0.385
<h4	0.4753	0.3682	0.3706	0.2918
18px;	0.7366	0.5424	0.5386	0.3862
font-weight:	1.0000	1	0.9945	0.9955
500;	1.0000	1	1	0.9958
</h4>	0.4232	0.2952	0.3411	0.2366
<span	1.0000	1	1	0.9954
	0.3113	0.2277	0.2036	0.2033
<div	0.9372	0.9295	0.9123	0.8569
grid;	1.0000	0.7893	0.4375	0
grid-template-columns:	1.0000	1	1	1
repeat(4,1fr);	1.0000	1	1	1
gap:	1.0000	1	0.9754	1
grid-template-rows:	0.7306	0	0	0
repeat(3,140px);	1.0000	1	1	1
	1.0000	0.9726	0.9728	0.9606
background-color:	0.4561	0.0878	0.1954	0.0513
#f5f5f5;	1.0000	1	0.9981	0.9918
grid-row:	1.0000	1	1	0.999
1;	0.6340	0.5828	0.6618	0.3294
column;	1.0000	1	1	0.9964
justify-content:	1.0000	1	1	0.999
space-around;	1.0000	1	1	1

grid-column:	0.7711	0.6819	0.6061	0.286
1/3;	0.4093	0.377	0.2201	0.0766
2;	0.6748	0.4779	0.4658	0.3246
3/5;	0.6042	0.4246	0	0.1613
3;	0.0769	0	0	0