CZ 4042

Neural Networks and Deep Learning

Project Report

SpaceNet 6 - Building Segmentation

John Lim Jin, U1822862E
Wong Yuh Sheng Reuben, U1820016H
Tan Chuan Xin, U1821755B

Prologue

Our project is on building segmentations of RGB images given by the Spacenet 6 challenge. We aim to create a deep neural net model to have the largest Intersection Over Union between our prediction mask and the ground truth mask. In this project, we explore various network architectures, loss and optimizer functions, dropout, and batch normalisation, upsampling methods as well as data augmentation to get the best test results.

Installation / Running Guide

In this project, we had modified source code from existing packages and repositories to expose low-level APIs that allow us to make changes to low-level building blocks in model architecture. Hence, those need to be custom-installed on top of repositories hosted on cloud platforms. To use our environment, please install from the provided environment.yml file.

Our environment has only been tested within the SCSE GPU cluster. We recommend logging into your account on the GPU cluster and performing the installation there. Follow the instructions on https://github.com/reubenwong97/spacenet_6/blob/master/README.md for a beuatified markdown version that is easier to read, or the instructions below (they are the same).

Installation
Clone the repository through git or unzip the uploaded zip file.

1. git clone https://github.com/reubenwong97/spacenet_6.git
2. conda env create -f environment.yml
3. conda activate tf2.1

Install our customised repos (Please install classification_models first):

1. git clone https://github.com/reubenwong97/models_dev.git
    a. Please clone this repository outside of the spacenet_6 folder
2. cd classification_models_dev
3. pip install .
4. cd ../segmentation_models_dev
5. pip install .

Download and obtain the same directory structure for our data. The data_project folder extracted containing test and train directories should be placed in the root directory of spacenet_6. The data can be downloaded here. Please refer to the below directory structure for greater clarity.

Running
To run scripts on SCSE's GPU server, please run from the project root directory which has the job.sh provided. As an example, to run the architecture_trial for resnet18, run the following command from project root in spacenet_6:

sbatch job.sh scripts/architecture_trial/architecture_trial_resnet18.py
<scripts> / <subfolder> / <filename>

Any outputs to std are redirected to the spacenet_6.out file.

Project Directory Structure

```
| (Project Root in SCSE GPU)
├───models_dev (custom repo to install - outside of spacenet_6)
│   ├───classification_models_dev
│   │   ├───classification_models_dev
│   │   │   ├───models
│   │   ├───tests
│   │   │   └───data
│   └───segmentation_models_dev
│       ├───docs
│       ├───images
│       ├───segmentation_models_dev
│       │   ├───backbones
│       │   ├───base
│       │   ├───models
│       ├───tests
└───spacenet_6 (spacenet_6 root directory)
    ├───archive
    ├───data_generation
    ├───data_project
    │   ├───test (SN_6_test.tfrecords in this folder)
    │   │   ├───img
    │   │   └───mask
    │   └───train (SN_6.tfrecords and SN_6_val.tfrecords in this folder)
    │       ├───img
    │       └───mask
    ├───results
    │   ├───checkpoints
    │   ├───figures
    │   ├───histories
    │   ├───predictions
    │   ├───sample_figs
    │   └───summary_figures
    ├───scripts
    │   ├───architecture_trial
    │   ├───data_augmentation
    │   ├───external_paramater_optimizer
    │   ├───external_parameter_decoderblocktype
    │   ├───external_parameter_decoderusebatchnorm
    │   ├───external_parameter_learningrate
    │   ├───external_parameter_loss
    │   ├───internal_parameter_activation
    │   ├───internal_parameter_decoderdroprate
    │   └───internal_parameter_decodernorm
    ├───utils
```

Introduction

Project Inspiration

We were inspired by the SpaceNet 6 - Multi Sensor All-Weather Mapping Challenge.
https://spacenet.ai/sn6-challenge/

SpaceNet 6 is one of the challenges in a series of geospatial machine learning challenges created by SpaceNet to accelerate the development of the field. The aim of SpaceNet 6 is to extract building footprints (image segmentation) using computer vision and artificial intelligence from Synthetic Aperture Radar (SAR) images captured by satellites. SAR images are different from optical images in that they utilise wavelengths that can penetrate cloud and vegetation cover, therefore providing a way to observe ground targets regardless of weather or terrain conditions.

Identifying building footprints through analysis of satellite imagery has many uses. In Humanitarian Aid and Disaster Relief (HADR), satellite images provide vital information about ground situations from a birds-eye view, when surface infrastructure has been destroyed and cannot be relied on to provide information. These updates are crucial for rescuers conducting their operations in guiding them towards buildings instead of building debris. There are countless day-to-day applications as well; Google maps comes to mind, where it identifies buildings around us. Military objectives like identification of target buildings, or concealed buildings can make use of building footprint analysis as well.

Therefore, we chose to work on the SpaceNet 6 dataset because of its potential, and our desire to work on something that can have real-world impact and be beneficial to people.

Our Project Idea

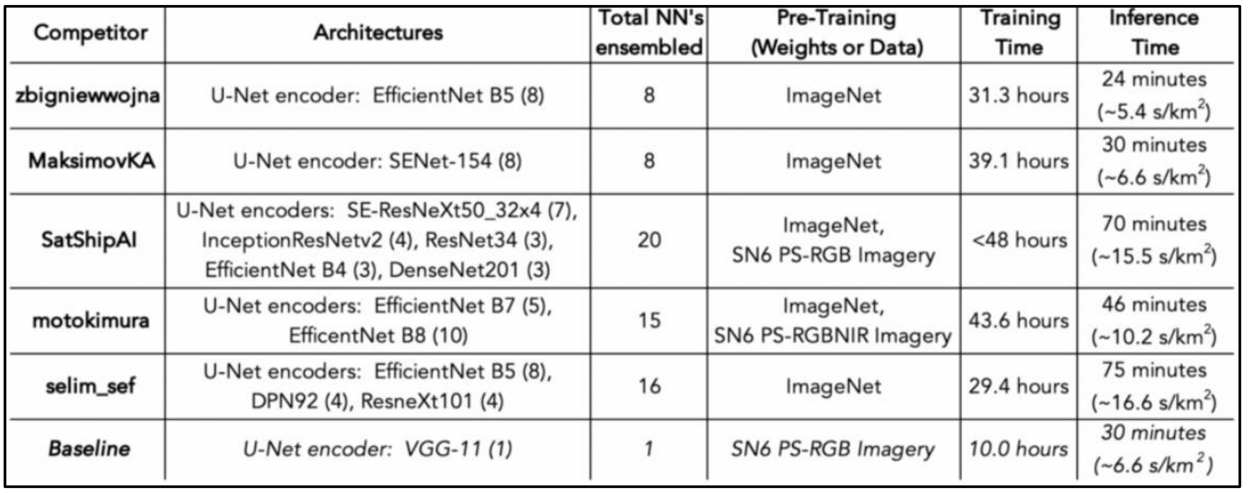
The SpaceNet 6 challenge introduces Synthetic Aperture Radar (SAR) images, a unique form of radar image that can be collected despite cloud cover and can be captured during the day or night, which makes it a valuable resource in situations as we mentioned earlier. SAR data has 4 channels and cannot be visualised as a traditional image.

SpaceNet 6 provides various image types as training data (PAN, PS-RGB, PS-RGBNIR, RGBNIR, SAR-Intensity), but only provides SAR-Intensity images as testing data. Refer to the file ***data_generation/1_data_understanding.ipynb*** to understand more about the SpaceNet 6 dataset. There is one image for each of the respective image types that corresponds to an identical geographical area. While the official challenge is to develop an image segmentation model utilizing SAR data, we decided against the use of SAR data for practical reasons:

1. SAR data is saved as float32, which causes the image to be much larger than RGB data that is saved as uint8. Out of the 80GB train data, SAR data comprised 41GB, and RGB data comprised only 7.71GB. Given our lack of persistent storage on the SCSE GPU cluster, we decided it was prudent to include RGB data only.

2. Machine learning on SAR is still a relatively new field, hence the challenge was created to accelerate the development of this field. However, this leads to a severe lack of resources to refer to when trying to utilise SAR data.

3. Best performing models from the challenge required the ensemble of many Neural Nets and required large amounts of training and inference time, which is impractical for our timeline and resources provided (the training times were on an average of 4 GPUs)



Competitor	Architectures	Total NN's ensembled	Pre-Training (Weights or Data)	Training Time	Inference Time
zbigniewwojna	U-Net encoder: EfficientNet B5 (8)	8	ImageNet	31.3 hours	24 minutes (~5.4 s/km^2)
MaksimovKA	U-Net encoder: SENet-154 (8)	8	ImageNet	39.1 hours	30 minutes (~6.6 s/km^2)
SatShipAI	U-Net encoders: SE-ResNeXt50_32x4 (7), InceptionResNetv2 (4), ResNet34 (3), EfficientNet B4 (3), DenseNet201 (3)	20	ImageNet, SN6 PS-RGB Imagery	<48 hours	70 minutes (~15.5 s/km^2)
motokimura	U-Net encoders: EfficientNet B7 (5), EfficentNet B8 (10)	15	ImageNet, SN6 PS-RGBNIR Imagery	43.6 hours	46 minutes (~10.2 s/km^2)
selim_sef	U-Net encoders: EfficientNet B5 (8), DPN92 (4), ResneXt101 (4)	16	ImageNet	29.4 hours	75 minutes (~16.6 s/km^2)
Baseline	*U-Net encoder: VGG-11 (1)*	*1*	*SN6 PS-RGB Imagery*	*10.0 hours*	*30 minutes (~6.6 s/km^2)*

Figure 1: Training and inference time of top models

For the above reasons, we decided against the use of SAR data, and instead focused on building segmentation using RGB images. For this purpose, we did some data-preprocessing to divide the train set they provided into a train-test split (as the original dataset did not have RGB data in its test set), which we will elaborate more on in our methodology section.

Therefore, we have settled on the following project idea:

Create an image segmentation model to extract building footprints from a PS-RGB image

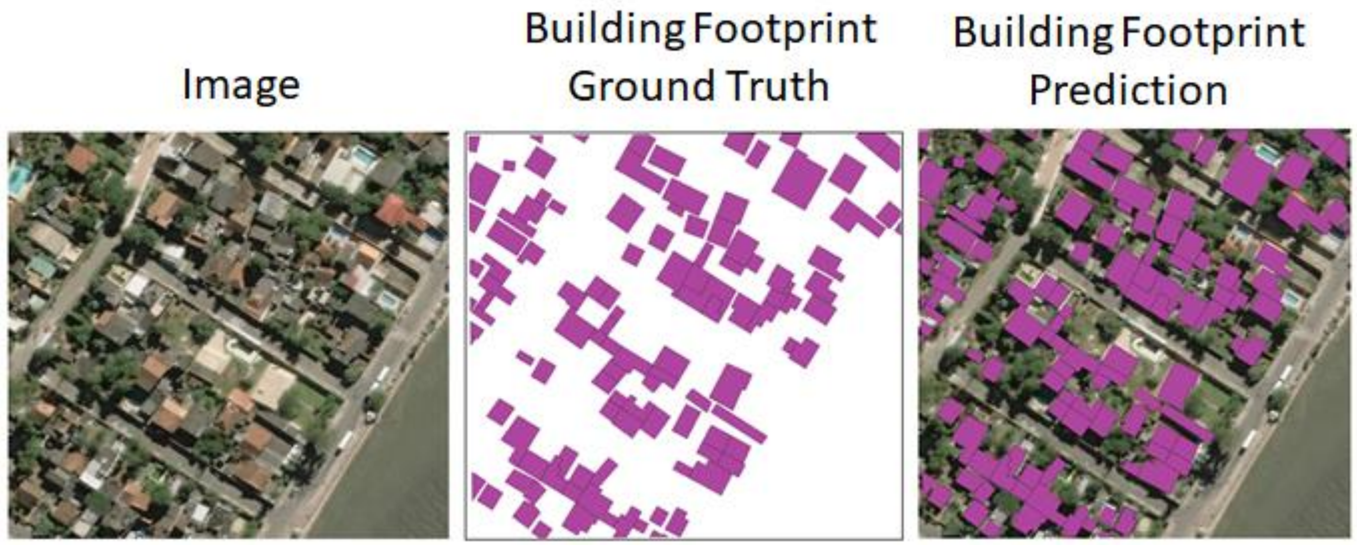


Image
Building Footprint Ground Truth
Building Footprint Prediction

Figure 2: Example of building segmentation

Existing Techniques

Unet Architecture

The goal of the challenge is to perform image segmentation on an image, to detect unique entities of a class (buildings). The top 5 performing models in the challenge all made use of Unet encoder-decoder frameworks. Unets are one of the most popular architectures for image segmentation problems.

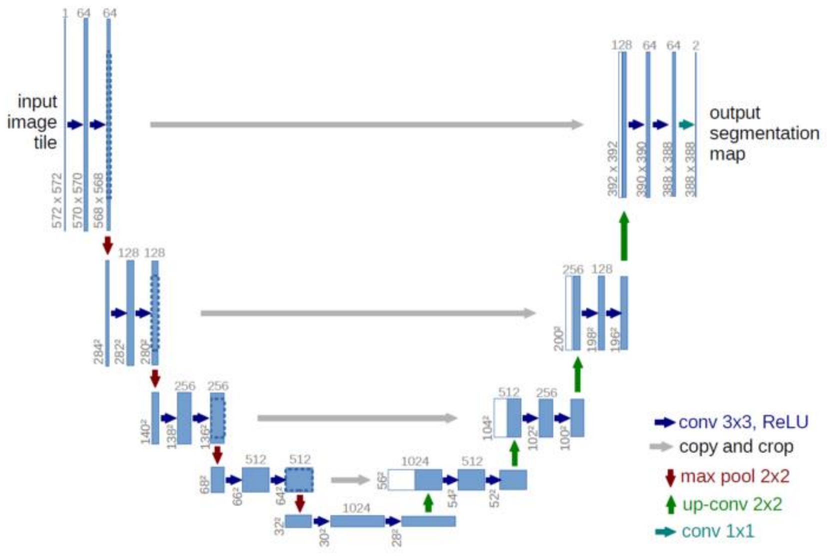


Figure 3: Unet architecture

Unets have 2 parts: a contraction path (encoder) that is made up of layers and blocks of convolutional and max pooling layers, and an expanding path symmetric to the contraction path (decoder), which also uses convolutional layers, however, it employs transposed convolution to make the images of a higher resolution. The encoder path captures context and important features while downsampling, while the decoder path upsamples the data without information loss and provides localization for the features learned in the encoder path, in order to perform pixel-segmentation to produce a prediction mask that is of the same resolution as the original image. As we can see in the above figure, there is an information highway from the encoder to the decoder. We concatenate the output from the decoder transposed convolution with the feature maps from the encoder at the same level, and this helps the transferring of fine-grained information from encoder to decoder. This helps to maintain fine details when the decoder upsamples and prevents a ‘patchy’ output image. Unets are considered a state-of-the-art framework for image segmentation.

Unet encoders can easily make use of classic deep neural networks. ResNets, VGG, Efficientnets can all form the backbone of the encoding path for the Unet architecture by removing their respective fully connected layers and joining the end of the neural network to the decoding path. This allows for Unets to be easily adaptable to all kinds of image segmentation tasks by choosing a domain specific encoding backbone. This allows for Unets to be extremely versatile for all kinds of tasks, and to leverage on existing neural networks improvements.

Imagenet Pretrained Weights

Majority of models utilise ImageNet pretrained weights as their initial weights. ImageNet is a large visual database designed for use in visual object recognition software and research. There are over 14 million annotated images for training. The weights that have been pre-trained on these pictures are effective because the database they have been trained on is large, and hence they have good generalisation, and are effective for transfer learning.

Transfer learning is the method of using knowledge gained in one model/problem and using it to solve a different but related problem. Though transfer learning usually takes place by using a pre-trained model on a different problem, we instead use ImageNet weights and apply it to whatever model we have chosen in order to 'transfer' the knowledge gained from the ImageNet classification to our specific problem, which also involves classification to a degree. Moreover, training our own weights from scratch would require a large amount of time that we cannot afford for this project.

qubvel/segmentation_models | **qubvel/**classification_models

https://github.com/qubvel/segmentation_models | https://github.com/qubvel/classification_models

We refer to the excellent repositories created by Pavel Yakubovskiy. It provides a keras wrapper over pre-trained segmentation models based on various popular backbones neural network architectures. It is one of the most popular packages for building segmentation models and provides helpful loss functions as well. Most of the podium finishers in the official SpaceNet 6 challenge first train their models using this package before fine-tuning it. **segmentation_models** provides the Unet architecture, and uses classification_models as a dependency, which provides the neural network model backbones such as ResNet, VGG19 etc. Therefore, we will first explore the options offered to us through these packages, and subsequently attempt to improve the neural network architectures in the **classification_models** package.

We have cloned the two repositories into our own project to allow us to make modifications to the source code. Our development repository has the following structure:

models_dev
classification_models_dev
segmentation_models_dev

Our Methodology

From our study of existing technologies, with the observation that Unet architectures performed the best for this task, the next step would be to systematically narrow down to the best off-the-shelf backbone. Thereafter we will perform a hyperparameter sweep on conventional parameters, and finally attempt to modify the model architecture itself to develop a better building segmentation model.

Dataset Construction

Three-way data pipeline

We have opted to utilise a three-way data pipeline, which involves splitting the original data into train-val-test sets. Our original data comes from the PS-RGB images of the SpaceNet 6 training data. The data we have is broken into <training, validation, testing> sets. The split was done through random indexes on the original data.

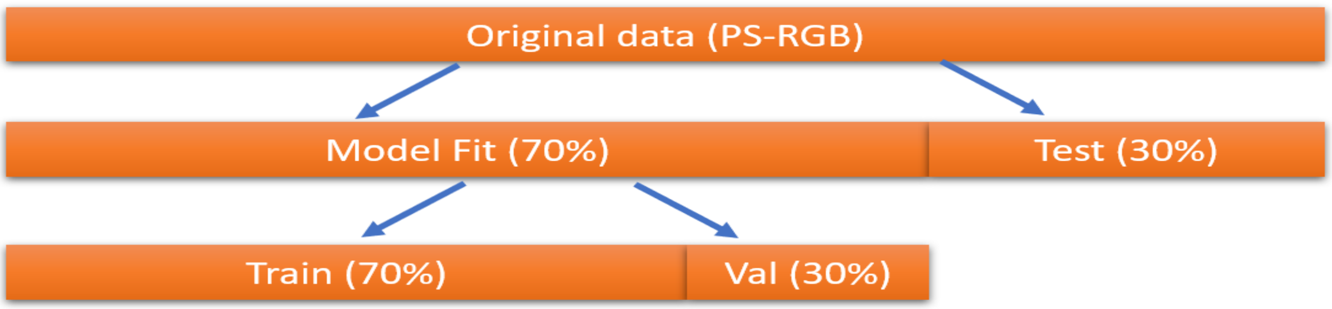


Figure 4: Three-way data split

The validation set is used together with the training set to calculate the entropy of a model and refine the model. The test set is held out and exclusively reserved for use as a final evaluation of the models' ability to generalize to unseen cases. The actual test data provided by SpaceNet 6 only contains SAR data, therefore we have a need to split out our own test set from the original training data for PS-RGB images.

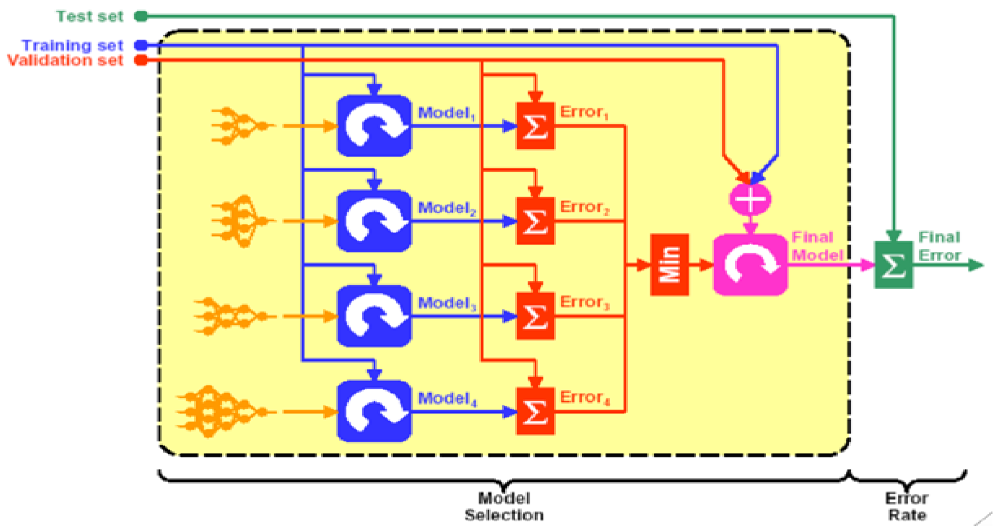


Figure 5: Application of three-way data pipeline

Our selection of the best model will be based on the maximum val_iou_score of the model during training. However, we will also evaluate every model trained on the test data to detect any possible discrepancies between our expected best model's performance on the validation set versus the test set.

Mask Generation

Please refer to Jupyter notebooks in the ***data_generation*** folder in the source code for detailed steps of our mask generation. The input images were read together with their corresponding geojson data from the ***data_all/AOI_11_Rotterdam/geojson_buildings*** folder. The geojson file contained polygon information that indicated the bounding boxes for the buildings present in the image. This allows us to draw a ground truth mask over the original image and create a binary mask for image segmentation.

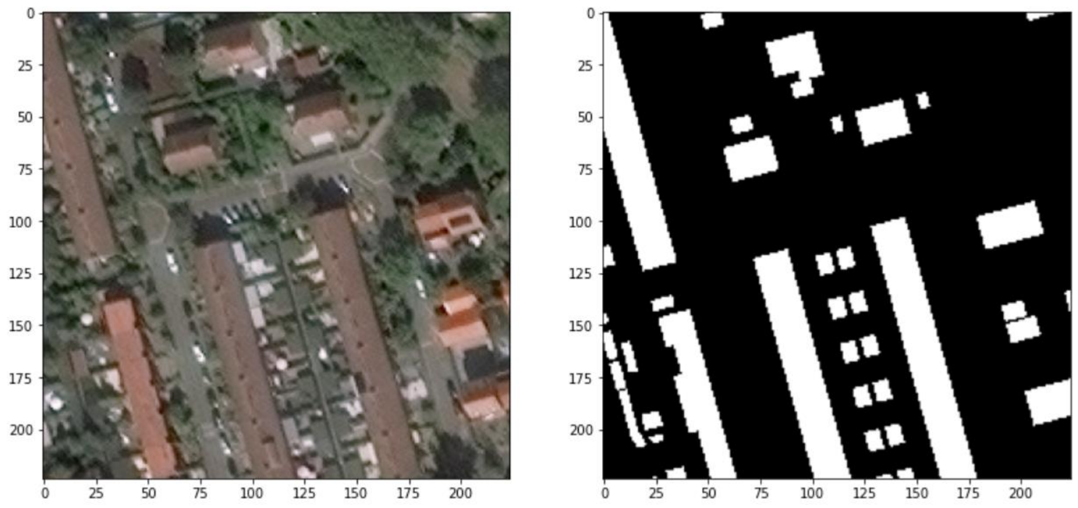


Figure 6: Binary mask for buildings in an image

The image and mask were both of dimensions 900x900 in height and width. However, most models accept images of smaller dimensions because their pretrained weights are all based on images of size 224x224, 226x226 etc. for practicality. To capitalize on transfer learning and pretrained models, this prompted us to generate input image sizes of 224x224. Reducing the image resolution is not recommended as it causes significant loss in image quality and detail. Hence, we tiled our input image and mask of 900x900 into sixteen different 224x224 tiles to suit this smaller dimension.

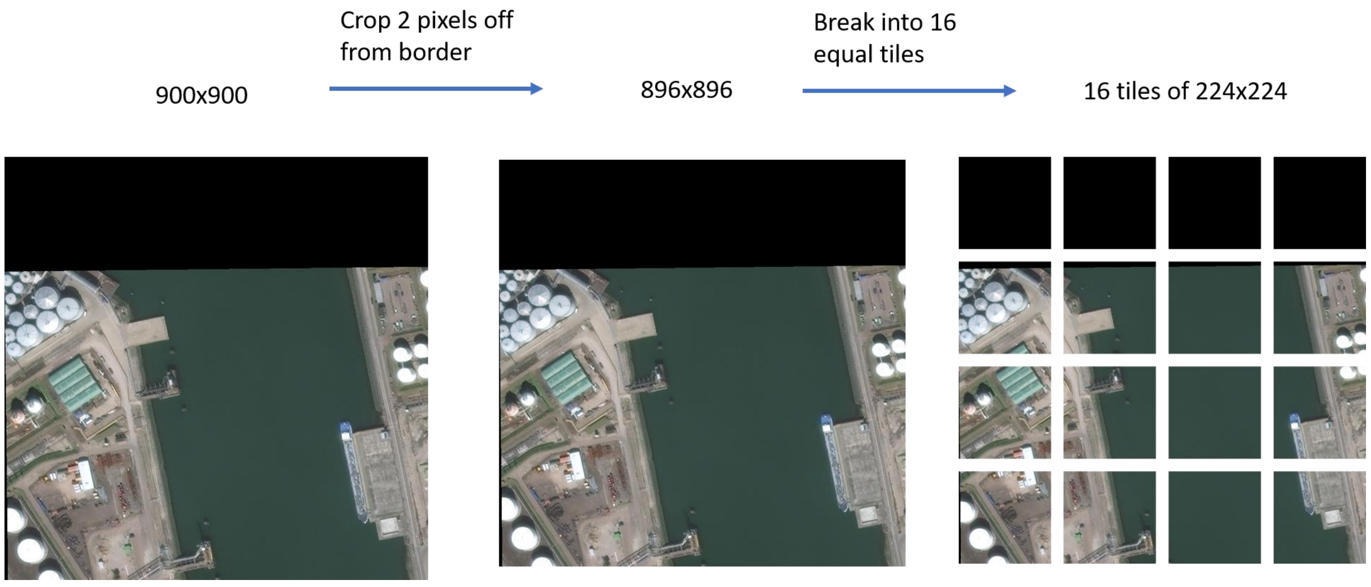


Figure 7: Tiling of image

The tiled images were then split into train-val-test sets through random indexes to attain a good distribution of our data.

Our tiled dataset totaled at 12.9GB. Given the SCSE GPU Cluster directory size limit, we opted to halve our dataset and compress by saving the image files as a. npy binary. This brought our full dataset to just over 5GB in size, which is much more manageable for the folder size limitations that we have.

The dataset is saved under ***data_project***.

Dataset Loading for Training

Bulk Data Loading

We first attempted the naive approach of loading the entire dataset into the SCSE GPU for training. However, this was a very slow approach and very memory inefficient because the entire dataset was over 5GB and had to be loaded into memory in one go. This resulted in an average data loading time of over 5 minutes, which made debugging and rapid testing very impractical. However, because all the data was available at once, it did make the training time per epoch much shorter.

```
def generate_train_test():
    paths = data_paths()
    data = [[], [], [], []]
    for index, path in tqdm(enumerate(paths), total=len(paths)):
        fnames = get_fnames(path)
        for fname in tqdm(fnames, total=len(fnames)):
            npy = rebuild_npy(path / fname)
            data[index].append(npy)
        data[index] = np.array(data[index])
    X_train, Y_train, X_test, Y_test = data[0], data[1], data[2], data[3]
    return (X_train, Y_train, X_test, Y_test)
```

Code Block 8: Bulk loading of train/test data

DataGenerator Class using keras.utils.Sequence

Data generators are very common with keras implementations over a tensorflow framework. This resolves the bulk data loading problem by only returning a batch size worth of data at any one time, thereby reducing memory requirements, and reducing time to start. While commonly used in many keras applications, it suffers from a drastic performance penalty. A key bottleneck in this approach occurs in the __getitem__ method which incurs a lot of I/O overheard by loading the .npy files one by one. The I/O cost outweighs the performance gains of loading in the entire dataset in one go, and in our testing, time per epoch increased by 16 times because of the persistent need for data fetching I/O operations. In fact, GPU utilization was frequently 0% as the model awaited data.

```
class DataGenerator(keras.utils.Sequence):
    def  __init__(self,  img_fname_list,  mask_fname_list,  img_path,  mask_path,  rebuild_func,  batch_size=128,
shuffle=True, test_gen=False):
        ...
        self.data_len = len(self.img_fname_list)
        self.on_epoch_end()
    def __len__(self):
        return int(np.floor(self.data_len) / self.batch_size)
    def __getitem__(self, index):
        indexes = self.indexes[index*self.batch_size:(index+1)*self.batch_size]
        img_list = [self.img_fname_list[k] for k in indexes]
        mask_list = [self.mask_fname_list[k] for k in indexes]
        X, Y = self.__data_generation(img_list, mask_list)
        if not self.test_gen: return X, Y
        else: return X
```

Code Block 9: keras.utils.Sequence data generator

Serialization into tf.data.Dataset Compatible TFRecords

Please refer to Python scripts in the ***data_generation*** folder in the source code.

Finally, we adopted tensorflow best practices and serialized our data into their protobuf format, tfrecords, for rapid loading of data, preventing our GPU from experiencing data starvation. After decoding the data, the Dataset object can prefetch data before computation such that they can be rapidly fed to the model. We see no degradation of performance in training, and in fact allows training to begin much more quickly than loading all the data into RAM. This allows us to quickly start training by reducing initial bulk loading overhead, and still benefit from the high data availability due to prefetching.

The drawback is the much more complex code required for loading data through tf.data.Dataset. To capitalise on the speed Dataset's offered, we had to serialize our rank 3 numpy tensors into 1 dimensional byte strings. These byte strings then had to be decoded and reshaped back into shapes of (224, 224, 3) for images and (224, 224, 1) for masks. Upon retrieving the Dataset object, we are finally able to map transformation functions to elements in our dataset, allowing us to augment images flowing through the pipeline, into our models.

```
def load_dataset(filenames, augment=False):
    ignore_order = tf.data.Options()
    ignore_order.experimental_deterministic = False
    dataset = tf.data.TFRecordDataset( filenames )
    dataset = dataset.with_options( ignore_order )
    dataset = dataset.map( decode_record, num_parallel_calls=tf.data.experimental.AUTOTUNE )
    if augment:
        dataset = dataset.map( data_augment, num_parallel_calls=tf.data.experimental.AUTOTUNE )
    return dataset
def get_dataset(filenames, batch_size=128, augment=False, epochs=100):
    dataset = load_dataset(filenames, augment=augment)
    dataset = dataset.repeat()
    dataset = dataset.shuffle(2048)
    dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
    dataset = dataset.batch(batch_size)
    return dataset
```

Code Block 10: TFRecords data generator

Our train and test tf.data.Dataset files were then saved under ***data_project***.

Performance Metric

This project used Intersection over Union (IOU) as our primary performance metric. Intuitively, IOU appears to be an extremely appropriate metric, since the segmentation task involves correctly predicting the segmentation mask of a given input image, and we would then like to quantify how well the ground truth and the predicted masks overlap. The IOU metric then measures the number of common pixels between the ground truth and the mask divided by the total number of pixels across both the ground truth and mask, giving us a good measure of the overlap between the 2 items. This is an obvious metric to select for our problem, given that we wish to have our predicted mask overlap as much as possible with the ground truth mask.

Conveniently, it also allows us to compare between models trained on various loss functions, which would otherwise be incomparable as the difference between their losses are arbitrary. Meanwhile, IOU represents a common basis with which we may evaluate which models will perform better on real-world images.

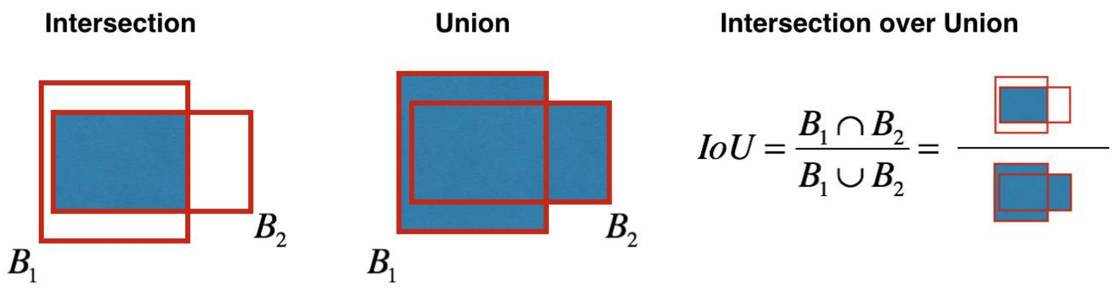
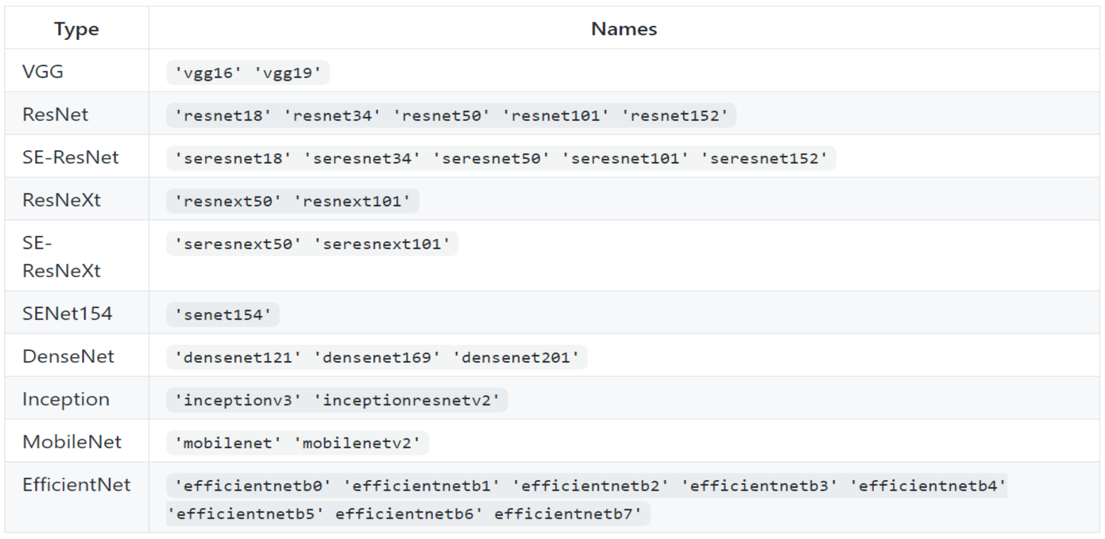


Figure 11: Visualization of Intersection Over Union (IOU)

Off-The-Shelf Backbones for Encoder

As alluded to in the Unet discussion, we can choose any deep neural network as a backbone model for the encoding of features from input images. For these backbone models, we employ transfer learning by providing the model with pretrained weights from the imagenet dataset, which has been shown to perform well for various tasks involving RGB images.

We can utilise several neural network backbones, as provided in the classification_models dependency used by the segmentation_models package. Using these industry-grade encoding backbones gives us a headstart by quickly creating a model that performs reasonably well, that we can then further improve on.



Type	Names
VGG	`'vgg16'` `'vgg19'`
ResNet	`'resnet18'` `'resnet34'` `'resnet50'` `'resnet101'` `'resnet152'`
SE-ResNet	`'seresnet18'` `'seresnet34'` `'seresnet50'` `'seresnet101'` `'seresnet152'`
ResNeXt	`'resnext50'` `'resnext101'`
SE-ResNeXt	`'seresnext50'` `'seresnext101'`
SENet154	`'senet154'`
DenseNet	`'densenet121'` `'densenet169'` `'densenet201'`
Inception	`'inceptionv3'` `'inceptionresnetv2'`
MobileNet	`'mobilenet'` `'mobilenetv2'`
EfficientNet	`'efficientnetb0'` `'efficientnetb1'` `'efficientnetb2'` `'efficientnetb3'` `'efficientnetb4'` `'efficientnetb5' efficientnetb6' efficientnetb7'`

Figure 12: Available neural networks from segmentation_models

We reference the work of SpaceNet 6 winners and will attempt to train a preliminary model on the following vanilla backbones and select the best backbone to carry out the rest of our experimentation on:

1. VGG variations
2. ResNet variations
3. SE-ResNeXt variations
4. Efficientnet variations
5. SENet154

External Hyperparameter Sweep

After selecting the best backbone for our encoder, we will then perform an external hyperparameter sweep for that backbone. This involves sweeping across various hyperparameters that can be tuned using the segmentation_models package API, or through keras' model training API. We will fine-tune our model on the following hyperparameters, which will be explained under the section **Experiments.**

1. Loss Function
2. Optimization Algorithm
3. Learning Rate
4. Decoder Block Type
5. Decoder Batch Normalization

We will sequentially optimize each of these hyperparameters and use the best model to feed into the next hyperparameter optimization, i.e. the best loss function will be used for all subsequent models when sweeping across optimization algorithms and learning rates. The hyperparameters have been selected based on their estimated impact on the performance on the model, e.g. giving priority to loss functions.

Notably, we are not sweeping across batch size and epochs. These are usually important hyperparameters to optimize as well. Changing the batch size will affect our gradient descent; small batch size tend toward stochastic gradient descent and offer a small regularizing effect due to the noise in frequently updating gradients for small sample sizes, while large batch sizes tend to be more stable but do not provide great accuracy due to gradient competition among many data samples in the batch. The number of epochs needs to be carefully chosen to ensure that we do not overfit on our data but are still able to achieve convergence during training.

We have opted not to do a hyperparameter sweep for batch size and epochs, and instead selected them based on practicality, as we are limited by our allocated RAM in the SCSE GPU Cluster, and also maximum run job run time. Our empirical findings indicated that batch_size=128 and epochs=100 offer a reasonable compromise between shorter run times of large batch sizes and gains in model performance using small batch sizes. Using this combination tends to generate run times of around 2 hours, which are also able to reach convergence for most models.

Internal Hyperparameter Sweep

The internal hyperparameter sweep is significantly more challenging than the external sweep. This involves tweaking the parameters that are not exposed by the backbone or Unet architecture API. Take for example a ResNet architecture, the skip connections within a residual block have already been carefully optimized and are the defining characteristic of the model architecture. The convolution operation is also dimension sensitive as it needs to maintain the correct input/output shape for the model to work and cannot be changed at will. Therefore, we must carefully select internal parameters to modify for both the encoder and decoder.

The parameters we change depends on the encoder and decoder that performed the best after our external hyperparameter sweep. To change these internal parameters, we have cloned the **segmentation_models** and **classification_models** repositories and created our own development versions. This allows us to have control over the internal parameters of the Unet architecture and backbone neural networks. We will use our own **segmentation_models_dev** and **classification_models_dev** packages as dependencies for this project.

Data Augmentation

Although we have a sizable training, validation and testing dataset, scarcity will always exist. The ideal model should have all the input data available in the world to train with and make perfect predictions, but this is impossible due to resource constraints. Data augmentation is a technique to generate new training data from existing training data. For images, this is typically done through image transformations, which takes a source image and applies various modifications to that image, such as flips, crops, rotations, colour changes etc.

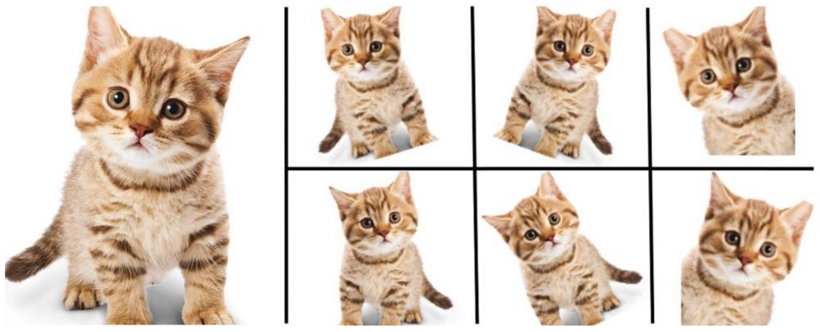


Figure 13: Simple example of data augmentation

Experiments

```
CheckpointCallback = ModelCheckpoint(str(PATH_CHECKPOINTS / (model_name + '.hdf5')), monitor='val_loss', verbose=1,
save_weights_only=True, save_best_only=True, mode='auto', period=1)
```

Code Block 14 : Checkpoint Callback

When running our experiments, we make use of a callback that will save the best model weights as a .hdf5 file upon completion of every epoch. This allows us to recreate our desired models that have the best performance, and not worry about fluctuations in both losses and IOU scores. The checkpointed model weights can also be reloaded into a recreated model to generate predictions on our test set.

The best hyperparameter at each experiment will be used for subsequent experiments.

We further utilise our script **histories_scores.py** to determine the best model for each experiment. We expect minute improvements between some models for certain experiments, hence a graph will be insufficiently precise to tell us which model performs better on the validation sets. This will be used to select the best model for each experiment. We also have another script **predictions_scores.py** to perform the same task for our test set, but as per the principle of three-way data split, this will simply be on a good-to-know basis.

Backbone Selection

In the first phase of our experiments, we focused our efforts in selecting the optimal architecture for our decoder from models that have been proven to perform while in various computer vision problems.

We initialized our Unet model with the following key hyperparameters:

1. Pretrained weights: ImageNet
2. Optimizer: Adam
3. Loss: BinaryFocalLoss(alpha=0.75, gamma=0.25)
4. Metrics: IOUScore
5. Epochs: 100
6. Batch size: 128

The main classes of encoders that were selected as the model backbone were referenced from the SPaceNet 6 competition winners:

1. VGG variations
2. ResNet variations
3. SE-ResNeXt variations
4. Efficientnet variations
5. SENet154

Model training was performed on SCSE’s V100 GPU with 32GB of RAM. However, even after allowing for GPU memory growth in our code, we were unable to be allocated more than 2.4GB of GPU RAM, as seen from Out-Of-Memory warnings given to us and abandoned the job. As such, we were unable to place larger models into the GPU RAM and were unable to experiment with them as a result.

Unfortunately, our experiments were thus restricted to measuring performances on a family of smaller ResNets, as they were the only ones that could successfully execute within our constraints of 2.4GB GPU RAM. ResNet versions that we could run were: ResNet18, ResNet34, ResNet50, ResNet101.

ResNets

In CNN, networks increase in performance as they get deeper. This is because they can fit a larger amount of data (increased generalisation) as there are more layers, because there are more parameters to learn and this gives the model flexibility. However, we notice that as the model gets deeper, the performance degrades, losing generalisation capability. One of the reasons why is the vanishing gradient problem: as backpropagation happens through the layers, the gradient is multiplied by the weight vector repeatedly, and easily shrink to zero over iterations of the chain rule. The result is that there is no updating of weights.

ResNets are short for Residual Networks and utilise skip connections that provide an 'information highway'. These skip connections usually skip over one block and provide residual input that is concatenated with the output from a layer. This concatenation of the residual helps the model learn an 'easier' function and provide a way for backpropagation of weights without having the vanishing gradient problem.

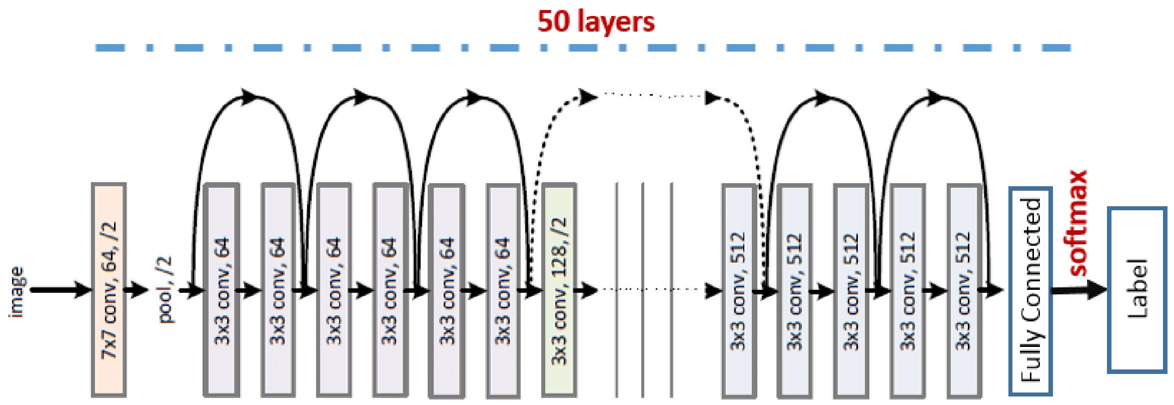
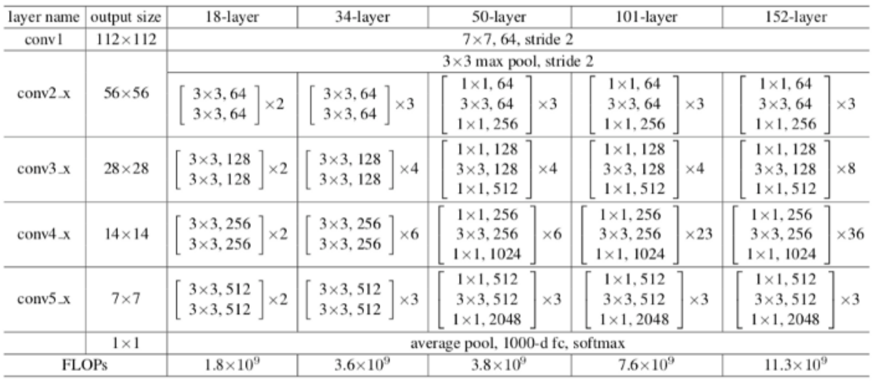


Figure 15: ResNet model skip connections



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 16: Sizes of inputs and number of feature maps at each block of ResNet

We can see that after the first 7x7 convolutional layer and the max pooling layer that is attached to it, there are 4 blocks of layers that behave similarly. Each of these blocks consist of 3x3 convolutional layers and bypass the input every 2 convolutions. Throughout the block, the width and height of the input remains the same (due to padding) and the number of feature maps remains the same.

The dotted line demarcates a change in the input volume between these blocks. In the previous image, we see there are dotted-line skip connections, and these demarcate changes in the input volume. In the above image, we see that between the blocks, the width and height of the input halves, but the number of channels/feature maps doubles to compensate for it. This reduction in width and height is done by the first convolutional layer in each block having a stride of 2 instead of 1 (throughout the block). Because the dotted line is a skip connection with different dimensions, we use the Projection Shortcut instead, which uses a 1x1 convolution with stride of 2, to create the input with the new dimensions, and we can concatenate this with the output of the next layer.

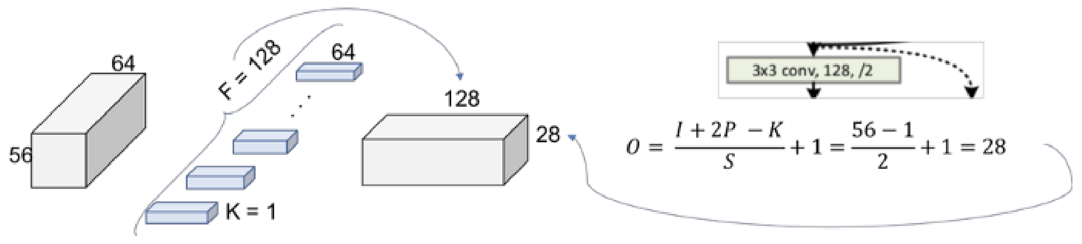


Figure 17: Projection shortcut

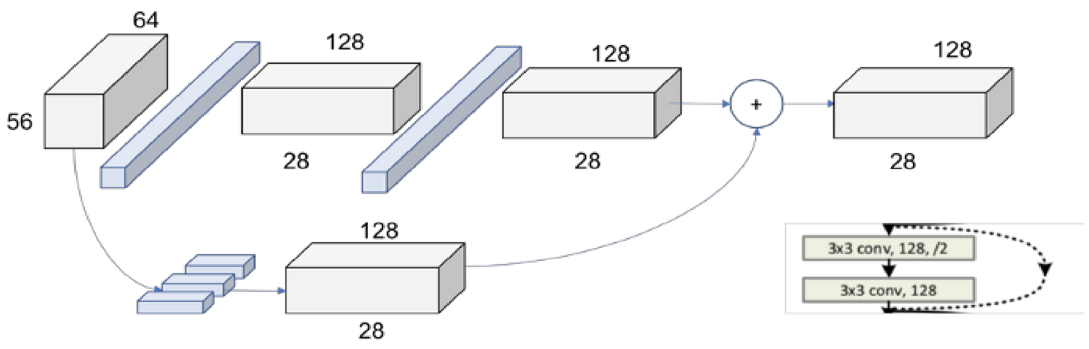


Figure 18: Dotted line skip connection between different output dimensions

This process is repeated throughout the blocks in ResNet.

Number of Layers	Number of Parameters
ResNet 18	11.174M
ResNet 34	21.282M
ResNet 50	23.521M
ResNet 101	42.513M
ResNet 152	58.157M

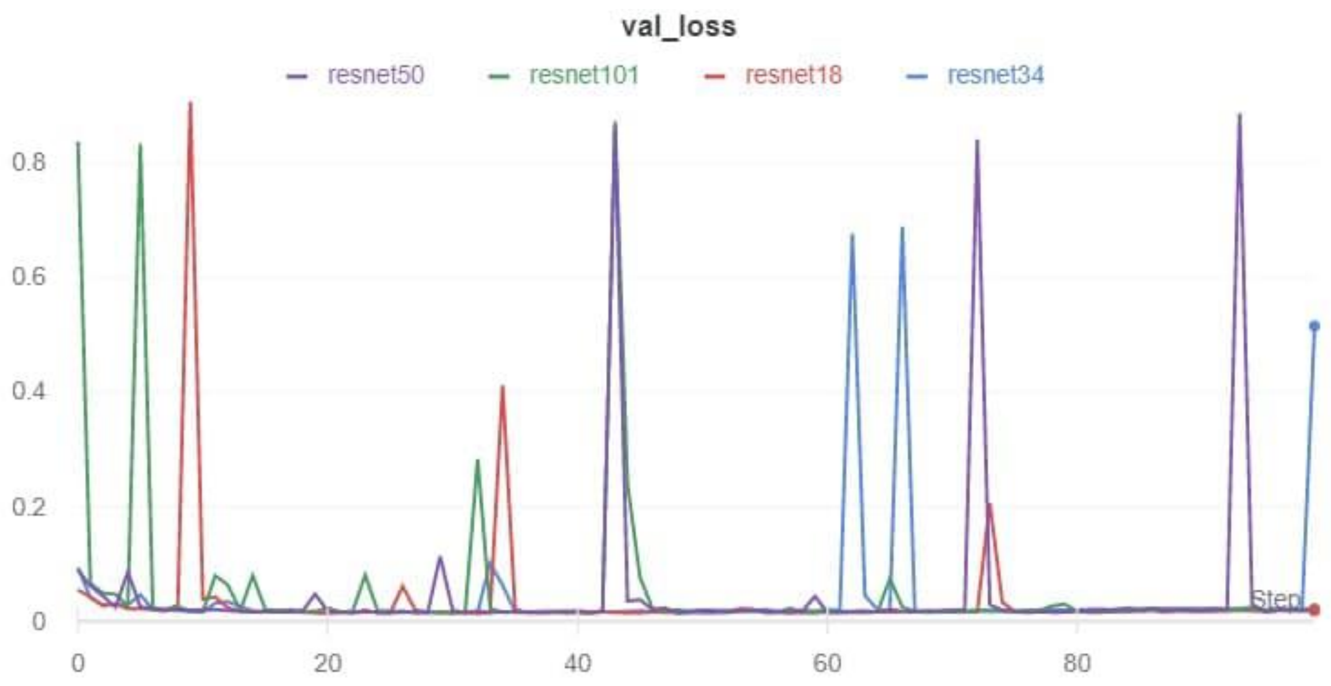
Figure 19: Number of parameters for different ResNet models

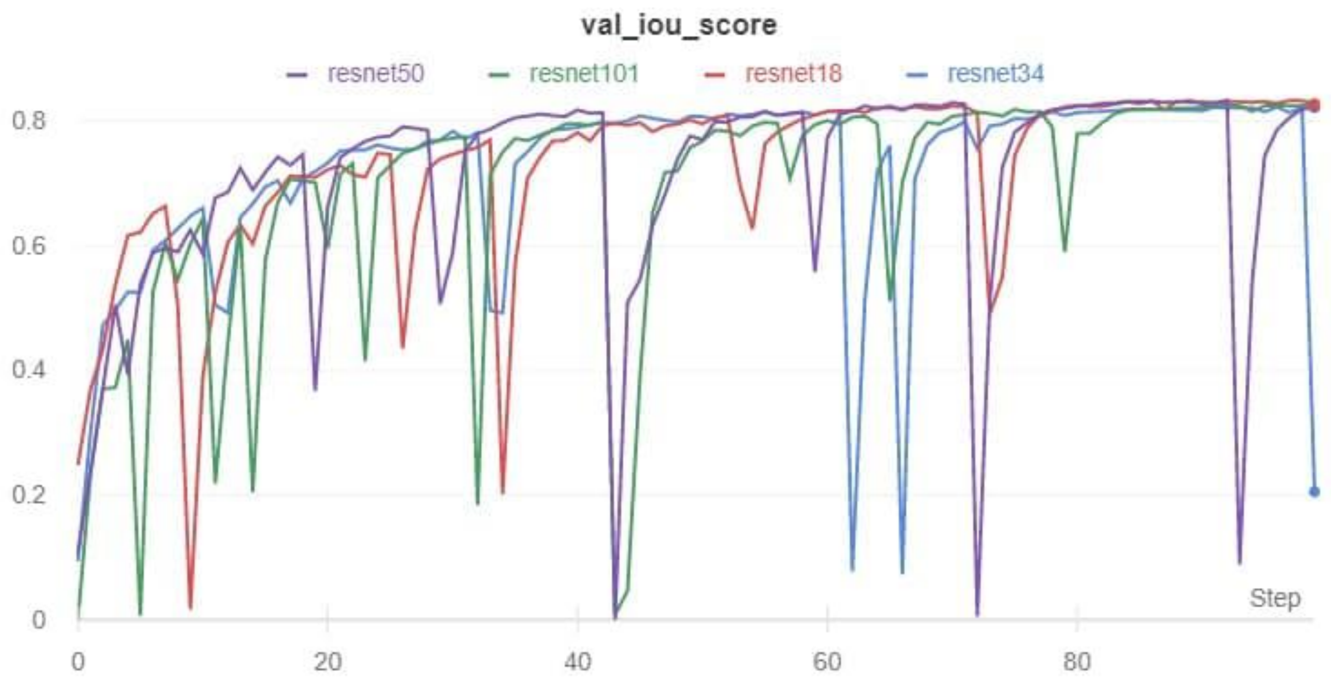
From the above image, we can see that the number of parameters needed to train the model increases as the number of layers increases. The difference between ResNet-34 and ResNet-50 is due to the composition of the blocks in these models. ResNet-50 has blocks with a [1x1 conv, 3x3 conv, 1x1 conv] compared to ResNet-34 which has blocks of [3x3 conv, 3x3 conv]. Due to the differing kernel sizes, the increase in 16 layers of ResNet-50 does not mean a large increase in the number of parameters from ResNet-34.

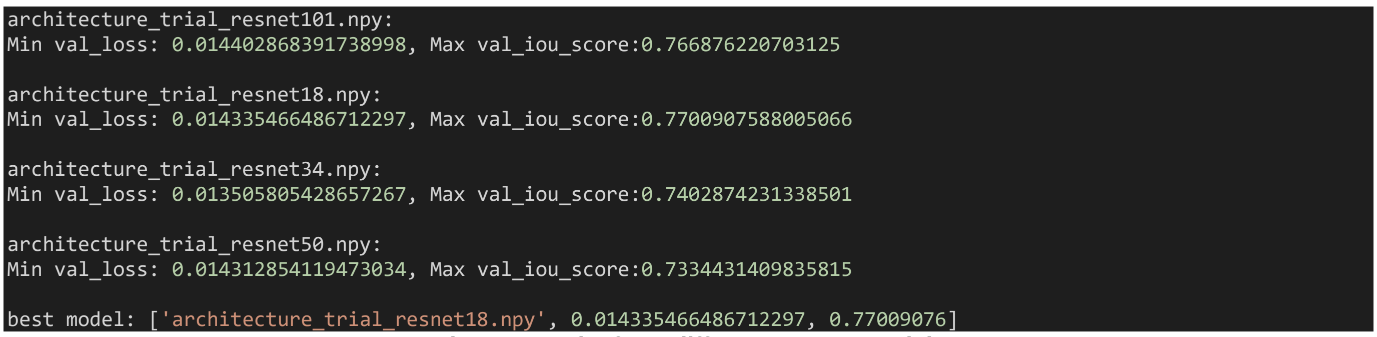
These amounts of parameters that are needed to be trained for the model directly impact our model choice, as we are both limited in terms of Server RAM as well as runtime.

Backbones tested = [ResNet18, ResNet34, ResNet50, ResNet101]

Results







```
architecture_trial_resnet101.npy:
Min val_loss: 0.014402868391738998, Max val_iou_score:0.766876220703125

architecture_trial_resnet18.npy:
Min val_loss: 0.014335466486712297, Max val_iou_score:0.7700907588005066

architecture_trial_resnet34.npy:
Min val_loss: 0.013505805428657267, Max val_iou_score:0.7402874231338501

architecture_trial_resnet50.npy:
Min val_loss: 0.014312854119473034, Max val_iou_score:0.7334431409835815

best model: ['architecture_trial_resnet18.npy', 0.014335466486712297, 0.77009076]
```

Fig 20: Results from different Resnet models

Discussion

Rather surprisingly, ResNet18 (the shallowest ResNet) outperformed all the other deeper ResNet models. This was an unexpected result as deeper models are supposed to be more effective at learning a greater number of characteristics and features, and that should technically translate into lower losses and higher IOU scores. Furthermore, one key feature of ResNets is the ability to avoid vanishing gradients due to the skip connections in place that allow for the input to be propagated further, and therefore deeper ResNets typically do not suffer from performance issues versus shallower models.

However, it should be noted that ResNet101 (deepest model we could test) only fell short by 0.033 versus ResNet18's val_iou_score. This is a very small difference, and it could simply be due to randomness of this training cycle that causes all the deeper models to perform worse.

Nonetheless, it is not appropriate to simply run the experiment repeatedly until our "expected result" is achieved. Therefore, we will select ResNet18 as the best model, and the backbone of choice for our Unet encoder for our subsequent experiments.

Loss Function - External Hyperparameter

Loss functions are one of the most important components of any neural network - it defines the cost for the model, and a good loss function will define an appropriate cost for the difference between actual and predicted results and allow the model to learn effectively. Therefore it is important to select a good loss function that is able to act as a definitive and accurate reflection of the efficacy of a given model in predicting desired outputs, as we optimize our model by seeking to reduce losses.

BinaryCELoss

Binary Cross Entropy Loss is useful in this case as we are segmenting the pixels in the image into 2 classes: either it belongs to a building or not.

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Fig 21: Formula for Binary Cross Entropy loss

For the formula, the loss rewards correct predictions of high confidence (probability), and penalises both correct predictions of low confidence, as well as wrong predictions of high confidence.

BinaryFocalLoss

Binary Focal Loss is a variation of Binary Cross Entropy. It weighs the contribution of each sample to the loss based on the classification error, i.e. it down-weights the contribution of easy examples (model predicted correctly with high probability/confidence), this enables the model to focus more on learning hard examples. Binary Focal loss also works well for highly imbalance class scenarios, for the reasons stated above, and it can be useful for us, because not every tile has a building in it, and on average, there are a lot more pixels that are not buildings in the ground truth compared to pixels which belong to buildings in the ground truth.

$$L(gt, pr) = -gt\alpha(1 - pr)^\gamma \log(pr) - (1 - gt)\alpha pr^\gamma \log(1 - pr)$$

Fig 22: Formula for Binary Focal loss

- **α** – Float or integer, the same as weighting factor in balanced cross entropy, default 0.25.
- **γ** – Float or integer, focusing parameter for modulating factor (1 - p), default 2.0.

The higher the value of gamma, the lower the loss for well-classified (easy) examples and extends the range for which an example receives low loss, making it more ‘focused’ on harder examples. When gamma equals to zero, it is equivalent to BCE loss. We experiment with different values of gamma to see which is the optimal amount of focus for the model on hard examples.

DiceLoss

The Dice coefficient is widely used in computer vision to calculate the similarity between 2 images and was adapted as a loss function to become Dice loss.

$$Dice = \frac{2\,|A \cap B|}{|A| + |B|}$$

Fig 23: Dice coefficient

$$L(precision, recall) = 1 - (1+\beta^2)\frac{precision \cdot recall}{\beta^2 \cdot precision + recall}$$

The formula in terms of *Type I* and *Type II* errors:

$$L(tp, fp, fn) = \frac{(1+\beta^2) \cdot tp}{(1+\beta^2) \cdot fp + \beta^2 \cdot fn + fp}$$

Fig 24: Dice loss

We can see the Dice loss has been adapted from the Dice coefficient, but the concept remains the same: find the overlap between the ground truth and the prediction. The Dice loss does it by calculating the loss from precision and recall, and the **β** hyperparameter is a float or integer to balance the precision and recall, and we experiment with different values of this parameter to find the optimal balance.

JaccardLoss

Jaccard loss is quite similar, it measures the Intersection Over Union (IOU) between the ground truth and the prediction as a measure of how well the model does. The Jaccard coefficient measures similarities between images, and is defined as below:

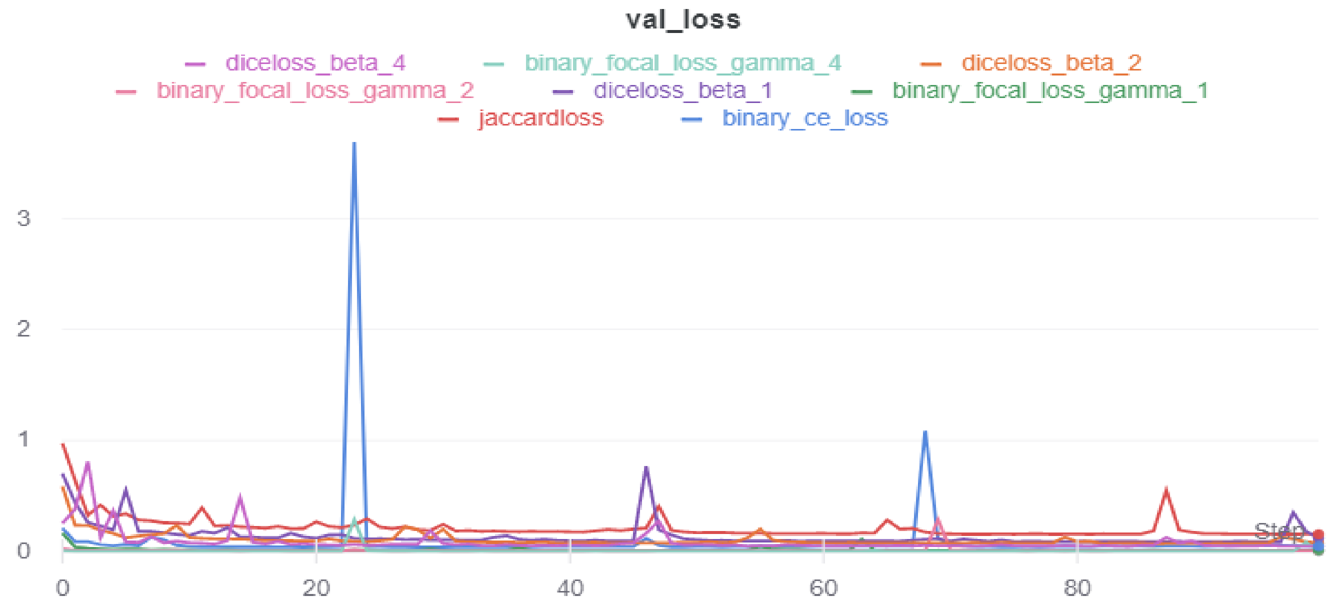
$$J(A, B) = \frac{A \cap B}{A \cup B}$$

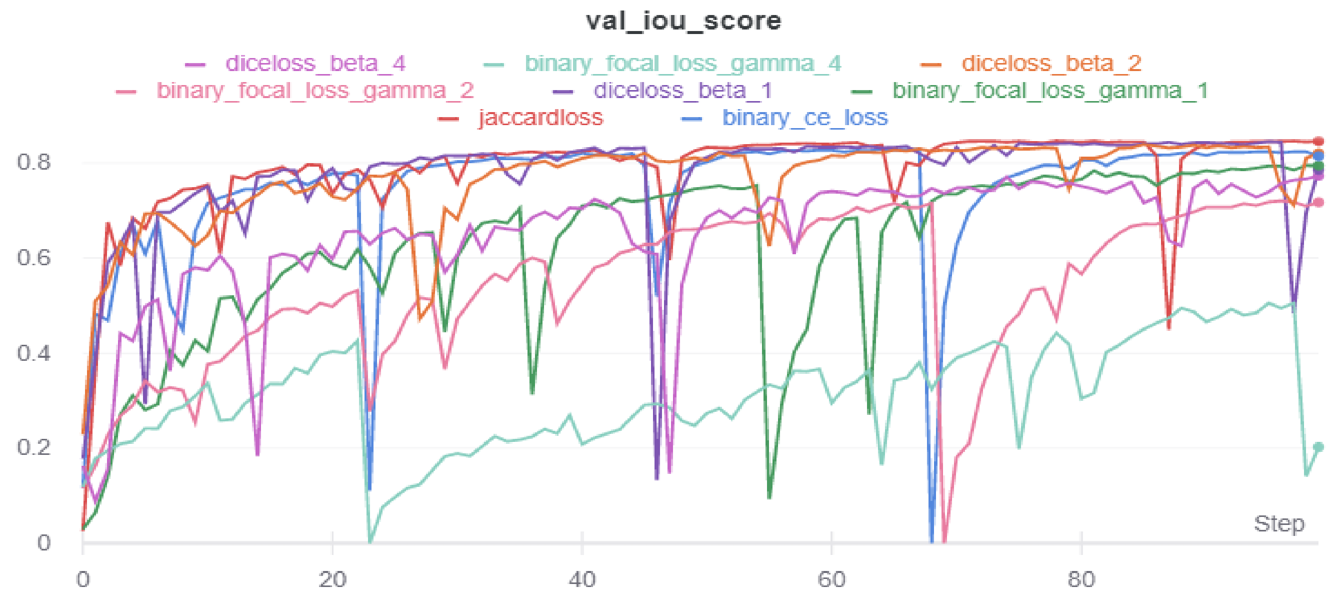
Fig 25: Jaccard loss

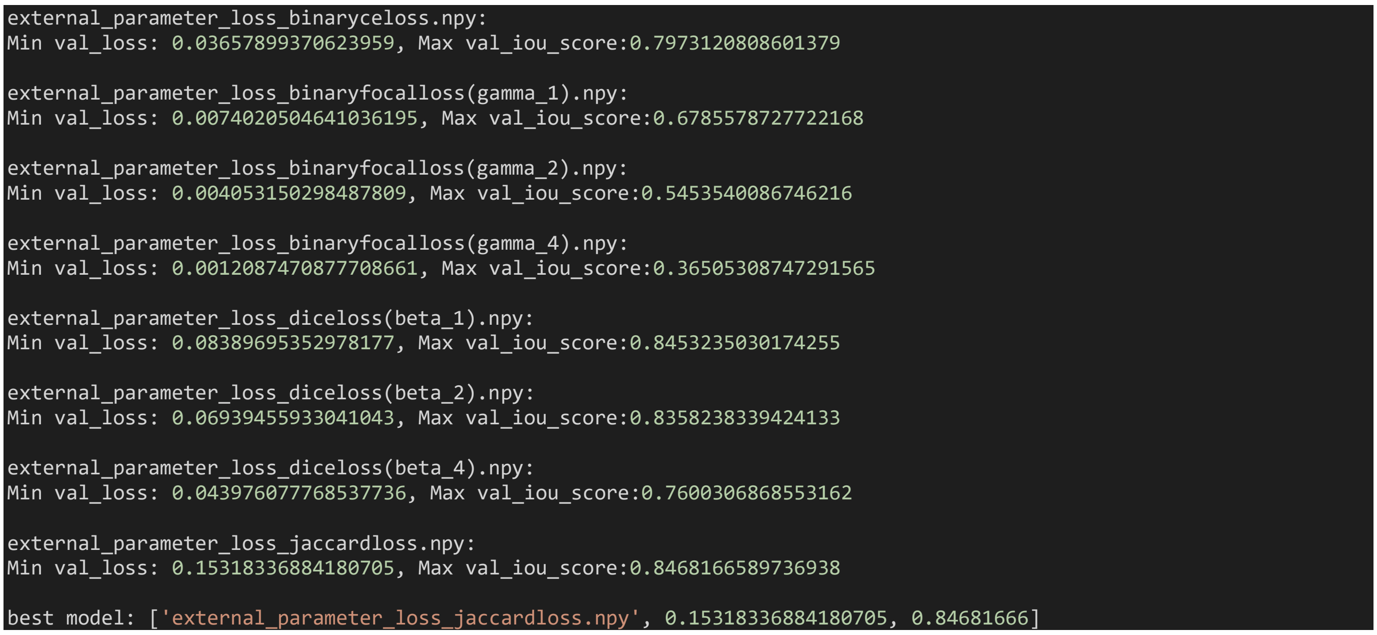
We implement Jaccard loss as one of our losses to experiment on because of the soundness of its principle regarding our image segmentation task.

Loss functions tested = [JaccardLoss, BinaryCELoss, DiceLoss(beta=1 | 2 | 4),
BinaryFocalLoss(gamma = 1 | 2 | 4)]

Results







```
external_parameter_loss_binaryceloss.npy:
Min val_loss: 0.03657899370623959, Max val_iou_score:0.7973120808601379

external_parameter_loss_binaryfocalloss(gamma_1).npy:
Min val_loss: 0.0074020504641036195, Max val_iou_score:0.6785578727722168

external_parameter_loss_binaryfocalloss(gamma_2).npy:
Min val_loss: 0.004053150298487809, Max val_iou_score:0.5453540086746216

external_parameter_loss_binaryfocalloss(gamma_4).npy:
Min val_loss: 0.0012087470877708661, Max val_iou_score:0.36505308747291565

external_parameter_loss_diceloss(beta_1).npy:
Min val_loss: 0.08389695352978177, Max val_iou_score:0.8453235030174255

external_parameter_loss_diceloss(beta_2).npy:
Min val_loss: 0.06939455933041043, Max val_iou_score:0.8358238339424133

external_parameter_loss_diceloss(beta_4).npy:
Min val_loss: 0.043976077768537736, Max val_iou_score:0.7600306868553162

external_parameter_loss_jaccardloss.npy:
Min val_loss: 0.15318336884180705, Max val_iou_score:0.8468166589736938

best model: ['external_parameter_loss_jaccardloss.npy', 0.15318336884180705, 0.84681666]
```

Fig 26: Results for various loss functions

Discussion

It is crucial to note that due to the differences in the way losses are computed for each loss function, a direct comparison of losses obtained is completely arbitrary and lacks a basis for comparison. Hence, when comparing the efficacy of a loss function, we compare the performance of each network trained on a particular loss function with a common, real-world performance reflecting metric, the validation IOU score.

Binary focal loss

Binary Focal loss is very similar to Binary Cross Entropy loss, but with an added emphasis on ‘harder’ examples. Gamma parameter in Binary Focal loss dictates the range at which the model considers an example to be ‘hard’, the higher the gamma, the more examples are classified as ‘easy’ and contribute less to the loss function. We can see that Binary Focal loss with $\boldsymbol{\gamma}$=1 performs quite well, but with higher gamma, performance drops drastically. This is likely because the higher gamma considers too many examples as ‘easy’, and the loss function receives little contribution from them. Effectively, that means the model is not learning parameters from ‘easy’ examples as much as it is from ‘harder’ examples, and at higher gamma, the model isn't learning important patterns and features in the data from the ‘easy’ set, where most test data would correspond to the ‘easy’ set. Hence, the model does not perform well when gamma is set too high.

JaccardLoss performed the best

Of all the metrics, JaccardLoss consistently performed well. JaccardLoss is a metric trying to maximise IOU, and our evaluation metric is IOU as well. Minimising JaccardLoss directly maximises IOU, and it is no surprise that JaccardLoss gives us the most consistent val_iou_score. IOU is commonly used as an evaluation metric in real world scenarios because it is a direct and clear comparison of ground truth vs prediction, and is especially applicable to image segmentation problems, where we need to create a prediction of the same resolution as the ground truth.

Optimization Algorithm - External Hyperparameter

Optimizers will affect how our model learns – choosing a bad optimizer will cause our model to either learn slowly, poorly, or never converge. A good optimizer can allow us to converge quickly and accurately. Different optimizers calculate gradients or learning rates in different ways.

Adagrad

Adagrad stands for Adaptive Gradient Algorithm. As the name suggests, it is an adaptive learning rate method, and adapts the learning rate to the parameter. The idea is to perform larger updates for infrequent parameters and perform smaller updates to frequent parameters.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \cdot g_t$$

G_t is sum of the squares of the past gradients w.r.t. to all parameters θ

Fig 27: Adagrad optimizer

Adagrad is useful in that it eliminates the need to manually tune the learning rate for parameters. In the denominator, we accumulate the sum of squares for past gradients. This results in the learning rate decreasing until it becomes too small. This is one of the main drawbacks of Adagrad, however, it can still be useful in certain situations, and we run our model once on Adagrad optimizer to see its effectiveness.

RMSProp

RMSProp is Root Mean Square Propagation. It tries to solve the diminishing learning rate problem in Adagrad by using a moving average of the squared gradient and utilises the magnitude of the recent gradient to normalise the gradient. The learning rate gets adjusted automatically and chooses a different learning rate for each parameter.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{(1-\gamma)g^2{}_{t-1} + \gamma g_t + \varepsilon}} \cdot g_t$$

Fig 28: RMSProp optimizer

γ is the decay term that takes a value from 0 to 1. **g_t** is the moving average of squared gradients. We can see that by using a moving average of past gradients instead of just accumulating them, the denominator does not keep increasing and the learning rate does not reduce to zero.

Adam

Adam is also known as Adaptive Moment Estimation. It functions as a sort of combination of Adagrad and RMSProp. It calculates the individual adaptive learning rate for each parameter from estimates of first and second moments of the gradients, while reducing the diminishing learning rates of Adagrad. It implements the exponential moving average of the gradients to scale the learning rate and keeps an exponentially decaying average of past gradients. It is one of the most popular gradient descent optimization algorithms because it is computationally efficient and has very little memory requirement. The exponential decay rates of the moving averages are controlled by **β**1 and **β**2 hyperparameters.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

m_t and v_t are estimates of first and second moment respectively

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{V_t}{1-\beta_2^t}$$

$\hat{m}_t$ and $\hat{v}_t$ are bias corrected estimates of first and second moment respectively

$$\theta_{t+1} = \theta_t - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$$

Fig 29: Adam optimizer parameter updating

SGD

SGD stands for Stochastic Gradient Descent. SGD is one of the simplest learning algorithms that we can use. In essence, we compute the gradient of the cost function with respect to the parameter that we are optimizing (usually the weights of the model) to discover the optimal set of values for the weights that minimizes our cost function. Stochastic Gradient Descent (formally) will update the weights with every training input. In our case, we implemented mini-batch gradient descent.

Mini-Batch Gradient Descent updates the weights after a specified number of inputs have been processed, termed as the batch size. This sped up the model, as SGD is very slow due to weights updating for every single training input.

$$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t}$$

Fig 30: Parameter updating with SGD

SGD-momentum

Gradient Descent aims to find an optimal set of weights that allows us to achieve the global/local minima for the cost function. However, these minimum points are often associated with steep gradient drops.

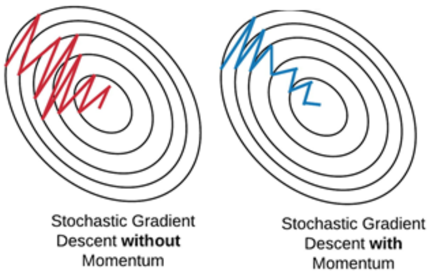

Stochastic Gradient Descent **without** Momentum
Stochastic Gradient Descent **with** Momentum

Fig 31: Effect of momentum

Vanilla SGD tends to oscillate across the slopes of these ‘ravines’ and take much longer to reach the minima. Therefore, the model will take a longer time to approach convergence at minima.

Momentum terms nudge the gradient towards the ravine. Essentially, the momentum term accelerates the gradient in the direction that previous gradients also imply. If the history of the past gradients has vectors that point in a certain direction, then there is momentum in that direction, and the momentum terms account for this direction, adding the momentum to the current gradient. Thus, if the direction of past gradients points toward the minima, the gradient will be accelerated towards the minima instead of taking small steps and bouncing across the ravine.

More formally,

γ momentum term	θ optimization parameter	η learning rate	v_t update vector at time t
$v_t=\gamma v_{t-1}+\eta\nabla_\theta J(\theta)$ the momentum from update vector at previous time (t-1) carries into the update vector at time t, and joins with actual gradient of loss function with respect to optimization parameter to form new update vector			

We expect that momentum terms will help navigate gradients better and converge faster and more accurately.

SGD-Nesterov

Nesterov acceleration optimization is very similar to momentum, but tries to avoid the shortcomings of momentum, which is that sometimes, the momentum updates the parameter too much;when the slope of the gradient changes in the opposite direction, momentum would update the parameters and overshoot where the slope of the gradient would have led. With Nesterov acceleration, we calculate gradient not with respect to the current step but with respect to the future step.

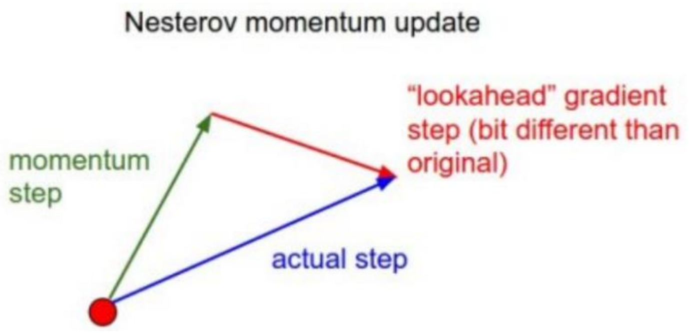


Fig 32: Nesterov momentum update

We use our momentum term to compute an approximation of the next position of our parameters.We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

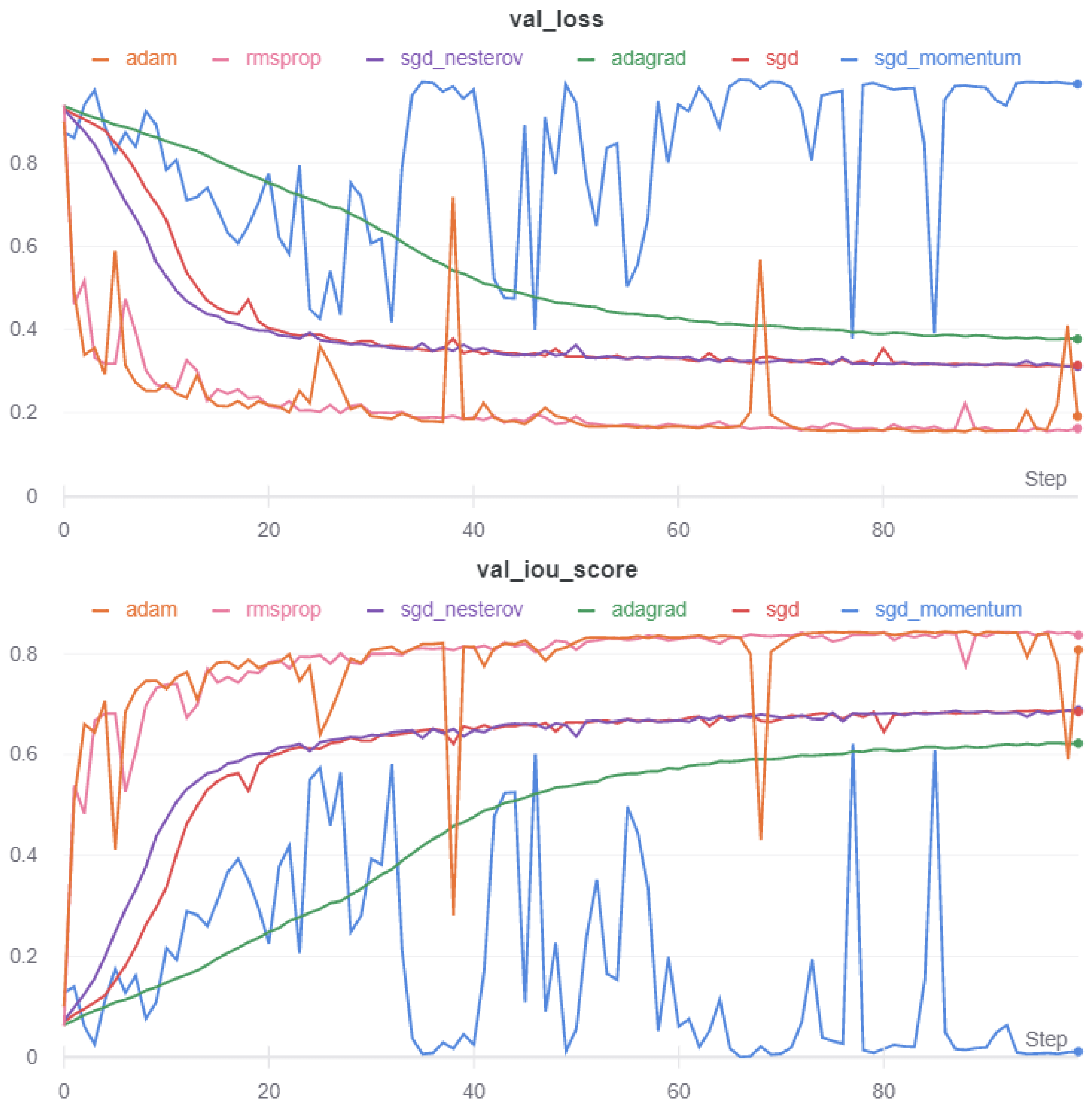
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

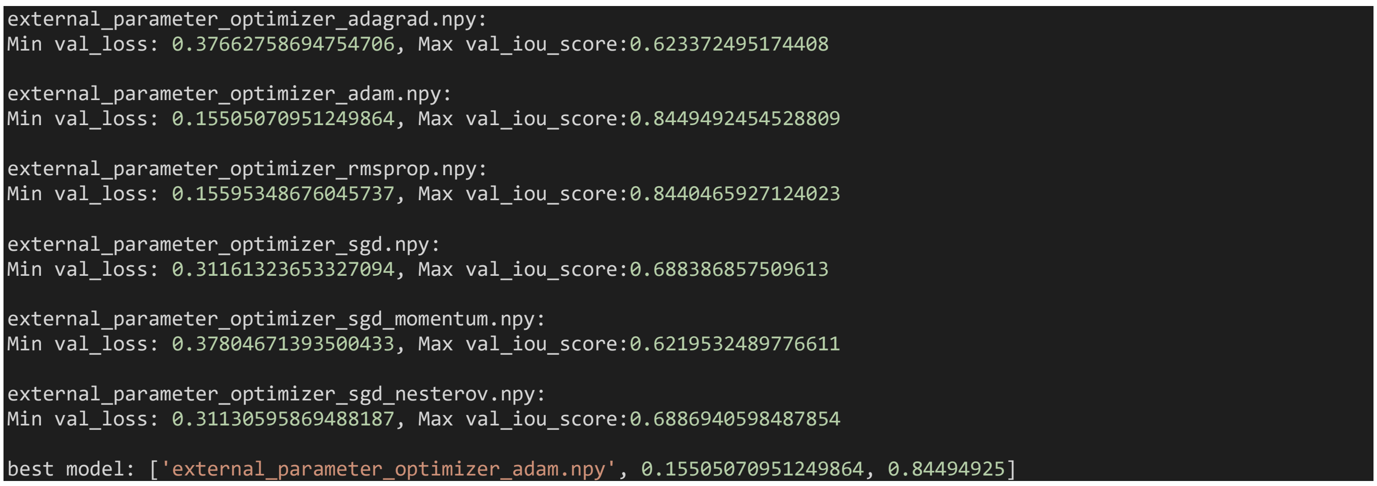
Fig 33: Nesterov updating of parameters

While momentum first computes the current gradient and then takes a big jump in the direction of the updated accumulated gradient , NAG first makes a big jump in the direction of the previous gradient, measures the gradient and then makes a correction. This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of some neural networks.

Optimizers tested = [Adam, RMSProp, SGD, SGD Momentum, SGD Nesterow, Adagrad]

Results





```
external_parameter_optimizer_adagrad.npy:
Min val_loss: 0.37662758694754706, Max val_iou_score:0.623372495174408

external_parameter_optimizer_adam.npy:
Min val_loss: 0.15505070951249864, Max val_iou_score:0.8449492454528809

external_parameter_optimizer_rmsprop.npy:
Min val_loss: 0.15595348676045737, Max val_iou_score:0.8440465927124023

external_parameter_optimizer_sgd.npy:
Min val_loss: 0.31161323653327094, Max val_iou_score:0.688386857509613

external_parameter_optimizer_sgd_momentum.npy:
Min val_loss: 0.37804671393500433, Max val_iou_score:0.6219532489776611

external_parameter_optimizer_sgd_nesterov.npy:
Min val_loss: 0.31130595869488187, Max val_iou_score:0.6886940598487854

best model: ['external_parameter_optimizer_adam.npy', 0.15505070951249864, 0.84494925]
```

Fig 34: Results for different optimizers

Discussion

SGD momentum performed poorly

Compared to all other models, SGD-momentum model performance fluctuates heavily, and underperforms all other models. This is representative of cases when we choose too high a momentum rate, that we update the weights by too much, it overshoots the optimum value, and because of the momentum term, it takes a long time for the updates to head towards the optimum once again. Momentum term also makes the performance of the model fluctuate widely, because we are taking too big a step when updating the parameters that the model cannot accurately determine the weights for the parameters. We can see that SGD without momentum does quite well, having performance gains at a steady rate. SGD-Nesterov, which is a more cautious form of SGD-momentum, converges quicker than SGD does at earlier epochs because of the momentum term, but because it tries to predict future steps and control the momentum, it does not suffer from the same issues as SGD-momentum does. Overall, however, as SGD converges to an optimum value, SGD-Nesterov also converges to that value, albeit a little faster. Momentum or Nesterov do not help the SGD model reach a new optimum value, it just helps in convergence speed in our case. For other cases, it can improve the optimal value if the graph has many local minimums with a global minimum, as without momentum, the parameters could converge to a local minimum.

Adam and RMSProp performed almost identically

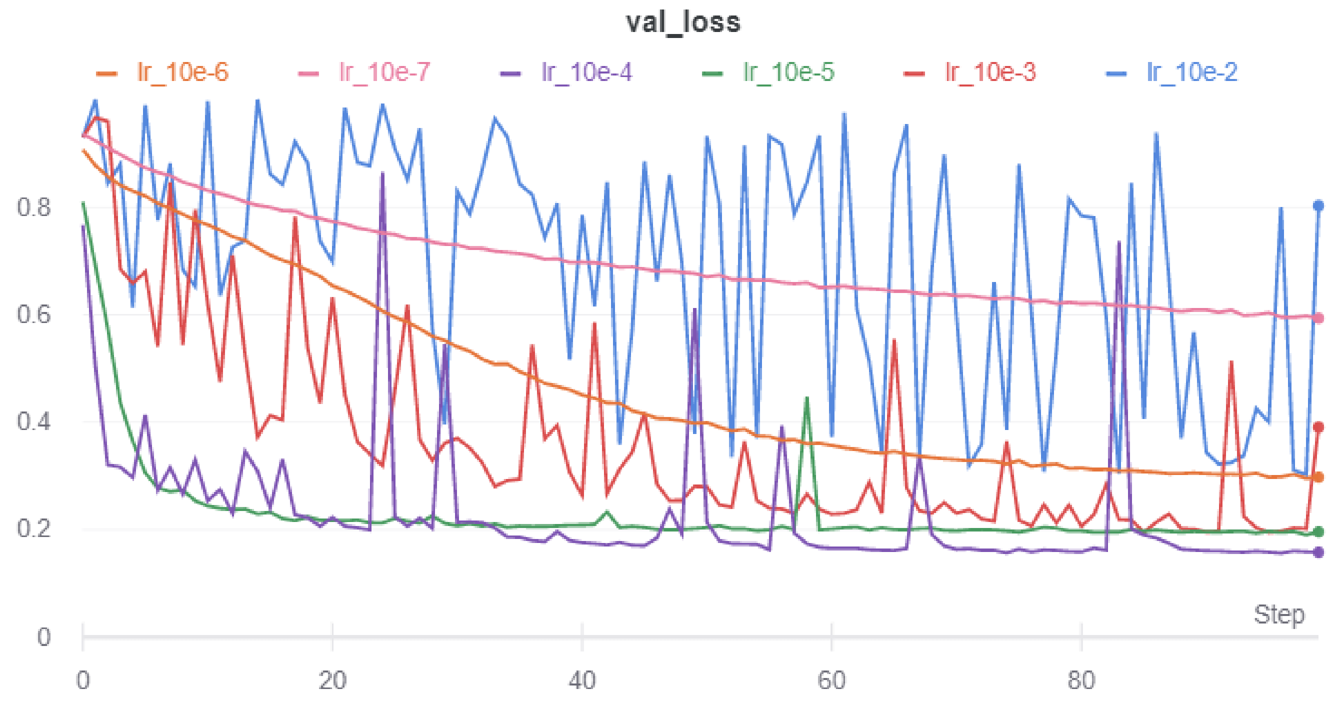
Both Adam and RMSProp are adaptive learning rate methods that seek to circumvent issues faced when using plain SGD, where learning rates need to be tuned extremely carefully. Thus, we see extremely rapid learning from both RMSProp and Adam, where they can obtain good performance with a small number of epochs trained. Although Adam’s performance has points of fluctuations, its overall performance is very similar to RMSProp. We note that RMSProp is highly consistent in both loss and iou_score metrics, but the highest val_iou_score obtained across all the models belonged to Adam. Therefore, we will stay true to our selection criteria and choose Adam as our optimizer of choice going forward.

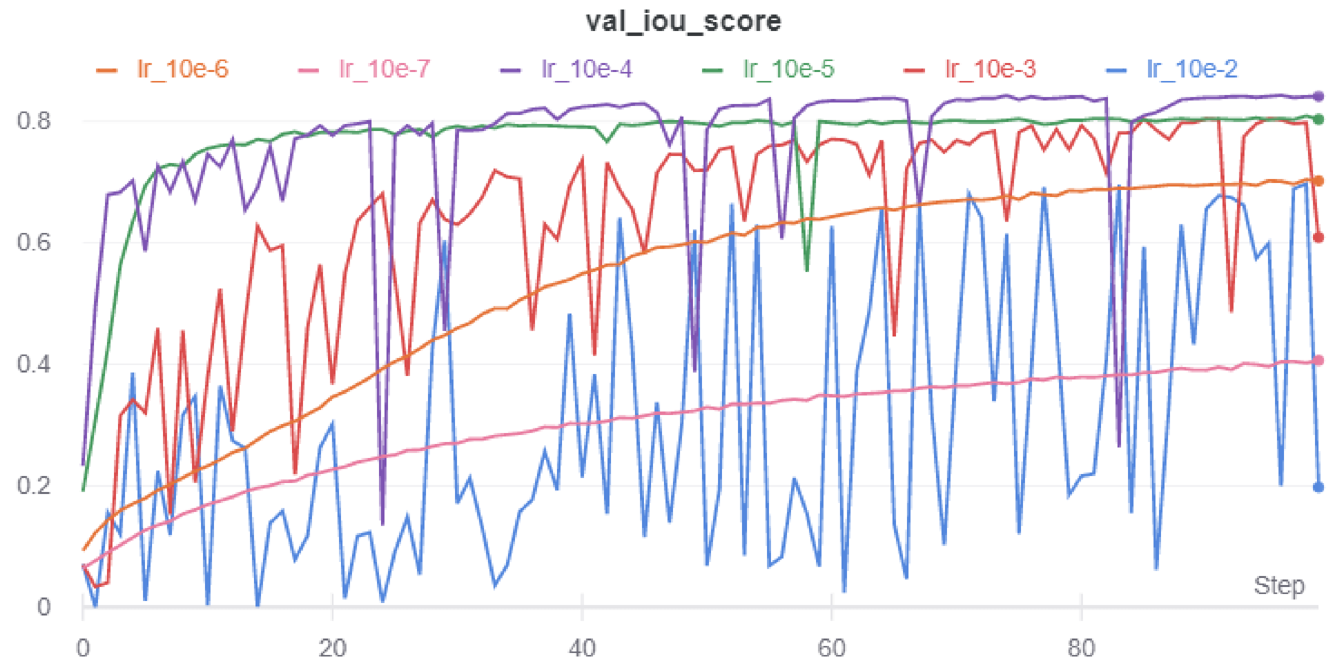
Learning Rate - External Hyperparameter

Learning rate is a hyperparameter that controls how much to change the parameters/weights in a model in response to the estimated error each time the model weights are updated. Choosing a learning rate that is too slow means our model converges to the optimum slowly or not at all and is computationally more expensive. Choosing too fast a learning rate can lead to both an unstable learning process as well as sub-optimal parameters. Choosing the right learning rate helps us reach our optimum parameters at a fast and stable speed. We iterate over a set of learning rates to find out which helps us to reach our optimum parameters within 100 epochs.

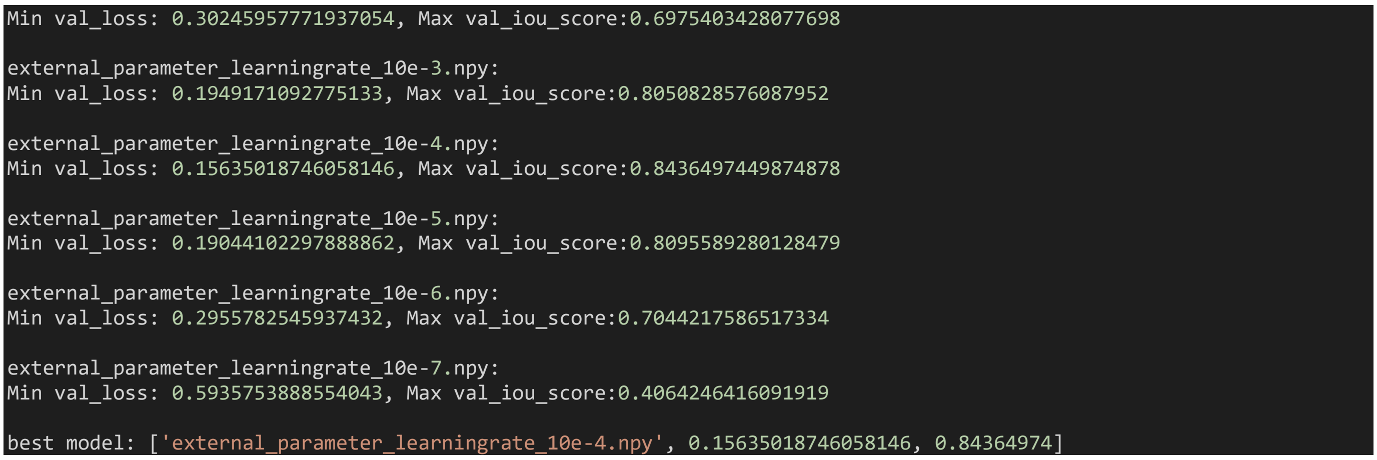
Learning rates tested = [10e^{-2}, 10e^{-3}, 10e^{-4}, 10e^{-5}, 10e^{-6}, 10e^{-7}]

Results





```
external_parameter_learningrate_10e-2.npy:
```



```
Min val_loss: 0.30245957771937054, Max val_iou_score:0.6975403428077698

external_parameter_learningrate_10e-3.npy:
Min val_loss: 0.1949171092775133, Max val_iou_score:0.8050828576087952

external_parameter_learningrate_10e-4.npy:
Min val_loss: 0.15635018746058146, Max val_iou_score:0.8436497449874878

external_parameter_learningrate_10e-5.npy:
Min val_loss: 0.19044102297888862, Max val_iou_score:0.8095589280128479

external_parameter_learningrate_10e-6.npy:
Min val_loss: 0.2955782545937432, Max val_iou_score:0.7044217586517334

external_parameter_learningrate_10e-7.npy:
Min val_loss: 0.5935753888554043, Max val_iou_score:0.4064246416091919

best model: ['external_parameter_learningrate_10e-4.npy', 0.15635018746058146, 0.84364974]
```

Fig 35: Results for various learning rates

Discussion

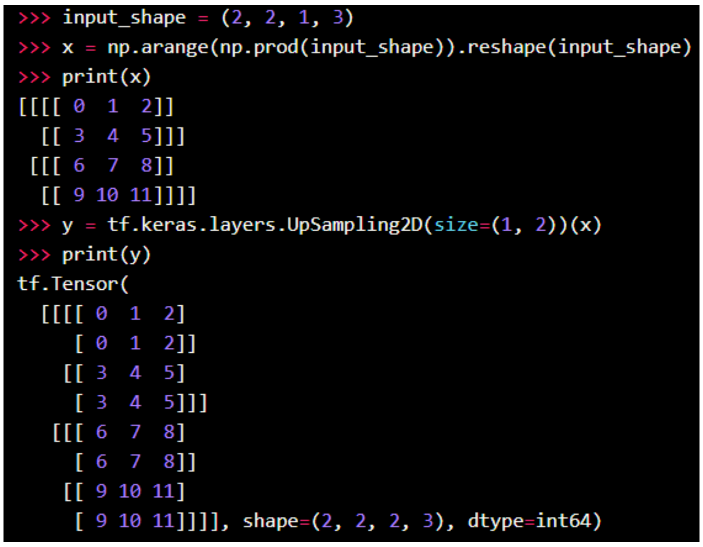
It is clear that learning_rate = 10e^{-4} performed the best. This is a moderate learning rate that is neither very large nor very small.

Decoder Block Type - External Hyperparameter

The decoder block type parameters/layers offered by the Keras API are Conv2DTranspose and Upsampling2D. The decoder block is responsible for upsampling the data into higher resolution images, and the 2 different layers go about doing this in different ways.

Upsampling 2D

The Upsampling2D layer multiplies each row and column by the integer or tuple passed into it.The Upsampling2D layer uses a nearest neighbour algorithm to fill in the new rows and columns, effectively multiplying the rows and column by the input parameter, as we can see from below.

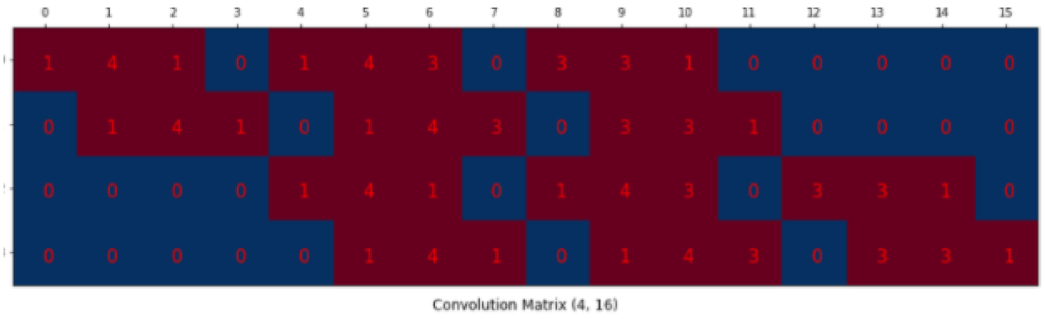


```
>>> input_shape = (2, 2, 1, 3)
>>> x = np.arange(np.prod(input_shape)).reshape(input_shape)
>>> print(x)
[[[[ 0  1  2]]
  [[ 3  4  5]]]
 [[[ 6  7  8]]
  [[ 9 10 11]]]]
>>> y = tf.keras.layers.UpSampling2D(size=(1, 2))(x)
>>> print(y)
tf.Tensor(
  [[[[ 0  1  2]
     [ 0  1  2]]
    [[ 3  4  5]
     [ 3  4  5]]]
   [[[ 6  7  8]
     [ 6  7  8]]
    [[ 9 10 11]
     [ 9 10 11]]]], shape=(2, 2, 2, 3), dtype=int64)
```

Fig 36 : Example of Upsampling2D layer

Conv2DTranspose

For Conv2DTranspose layer, the need for transposed convolutions arises from wanting to have a transformation in the opposite direction of a normal convolution while upsampling, i.e. we want to maintain the many-to-one relationship during convolution but in the opposite direction.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	4	1	0	1	4	3	0	3	3	1	0	0	0	0	0
0	1	4	1	0	1	4	3	0	3	3	1	0	0	0	0
0	0	0	0	1	4	1	0	1	4	3	0	3	3	1	0
0	0	0	0	0	1	4	1	0	1	4	3	0	3	3	1

Convolution Matrix (4, 16)

Fig 37: Transposed convolution matrix

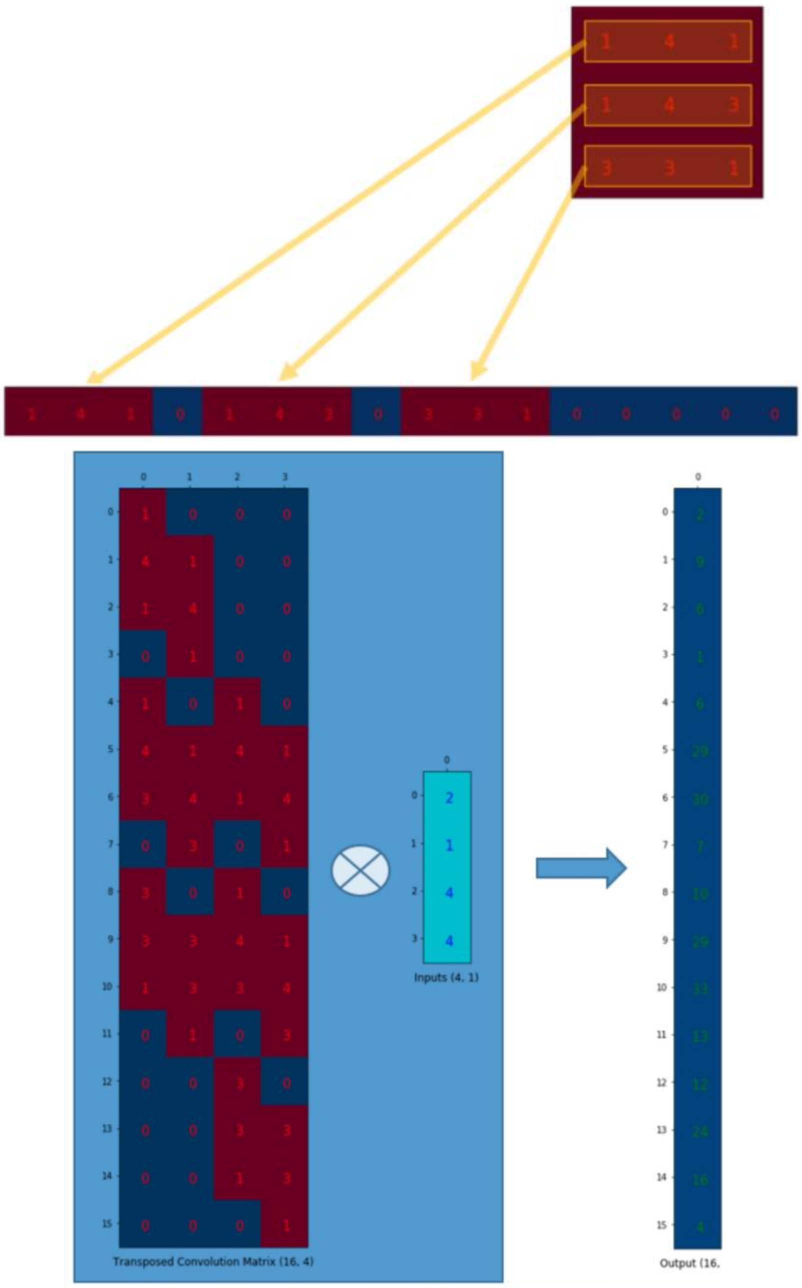
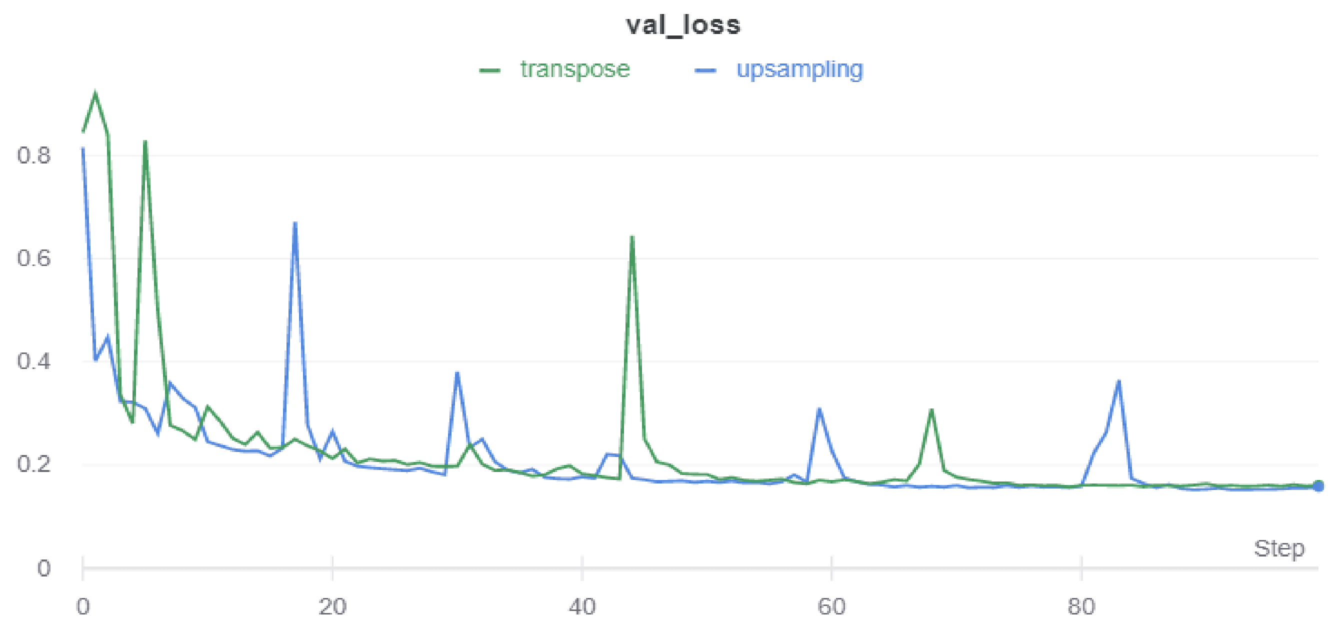


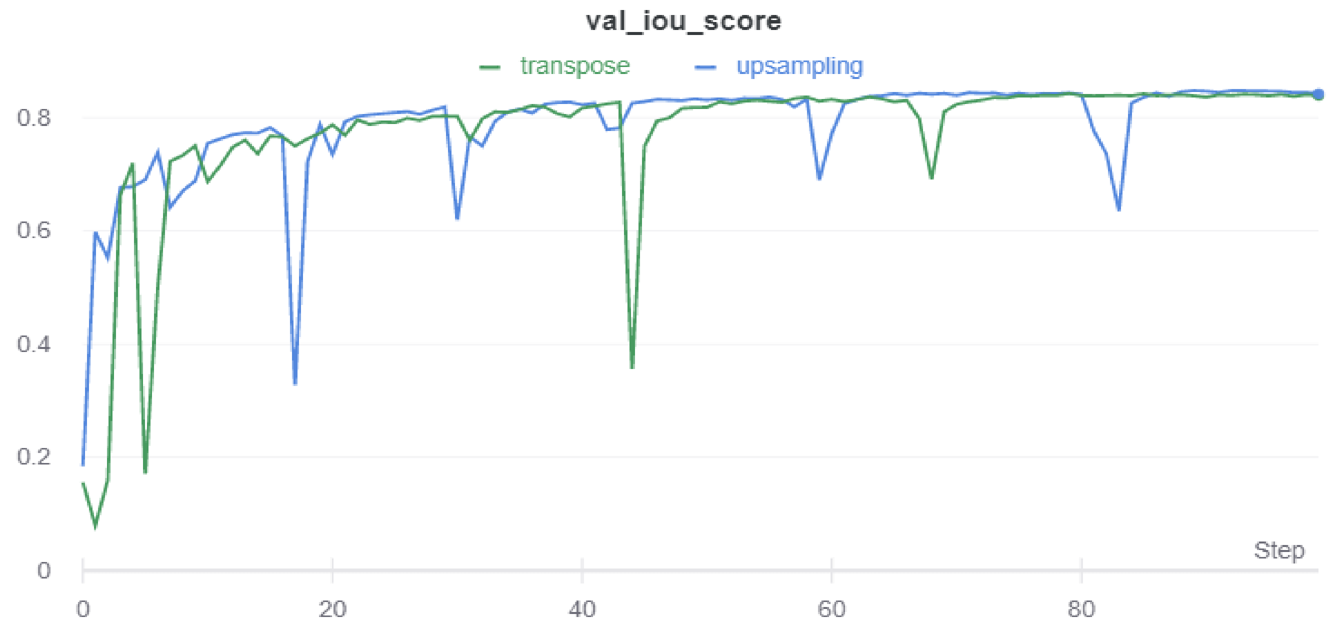
Fig 38: Transposed convolution

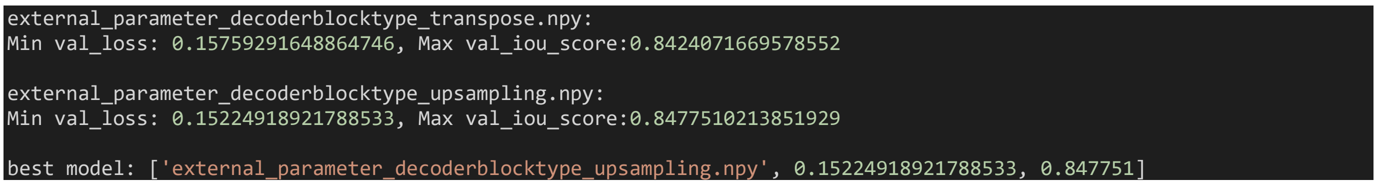
In the figure above, in the original Conv2D layer in the encoder (downsampling path), there is a 3x3 filter applied on an input to create a 2x2 matrix. In the Conv2DTranspose layer, we create a 4x16 matrix from a 3x3 filter (same size but different learned weights) by laying out the 3x3 matrix in each row, with different starting points. Then, we transpose it to create a 16x4 matrix, and flatten our 2x2 input into a 4x1 vector. Now, we can perform matrix multiplication to create a 16x1 vector, which we can then reshape into a 4x4 matrix, successfully upsampling a 2x2 matrix into a 4x4 matrix while maintaining a one-to-many relationship, opposite of the many-to-one relationship during convolution.

Decoder block types tested = [Upsampling, Transpose]

Results







```
external_parameter_decoderblocktype_transpose.npy:
Min val_loss: 0.15759291648864746, Max val_iou_score:0.8424071669578552

external_parameter_decoderblocktype_upsampling.npy:
Min val_loss: 0.15224918921788533, Max val_iou_score:0.8477510213851929

best model: ['external_parameter_decoderblocktype_upsampling.npy', 0.15224918921788533, 0.847751]
```

Fig 39: Results for transpose convolution against upsampling layer

Discussion

From the graph, we see that the performance of both blocks is very similar, albeit each model having performance dips at different epochs. Based on the theory of both upsampling layers, and now with empirical evidence, we can see that both layers are able to upsample data while filling in finer details and producing high-resolution images and maintain details from the original image. We conclude that both layers are viable as upsampling methods, but for the sake of this use case, we pick the upsampling layer that performs marginally better: Upsampling2D.

Decoder Batch Normalization- External Hyperparameter

Batch Normalisation

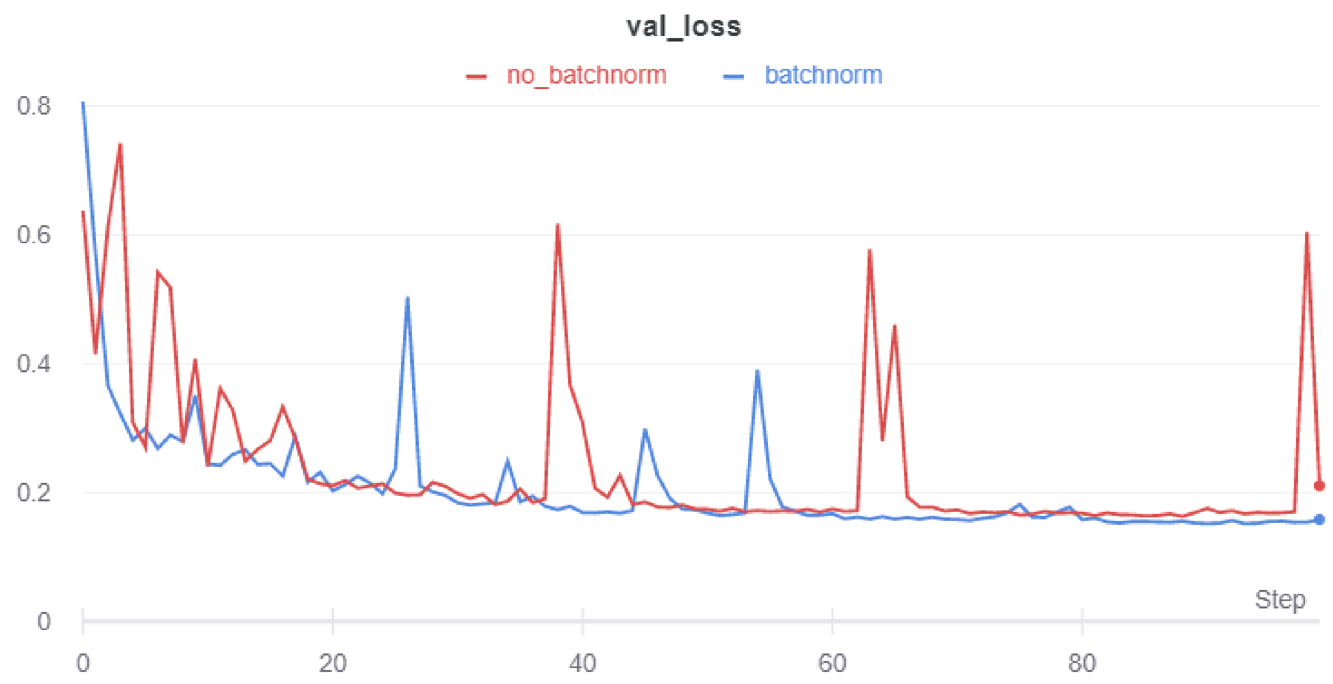
Training deep networks with tens of layers is challenging as they can be sensitive to initial random weights and configuration of the learning algorithm. One reason for this is that during backpropagation, weights of a layer are updated, assuming that the weights of the layers, and thus the resulting distribution of inputs to the layer, are fixed. However, because the earlier layers' weights get updated afterward, the distribution of inputs to the layer also can change. This can cause the learning algorithm to forever chase a moving target, and this effect is termed as 'internal covariate shift'.

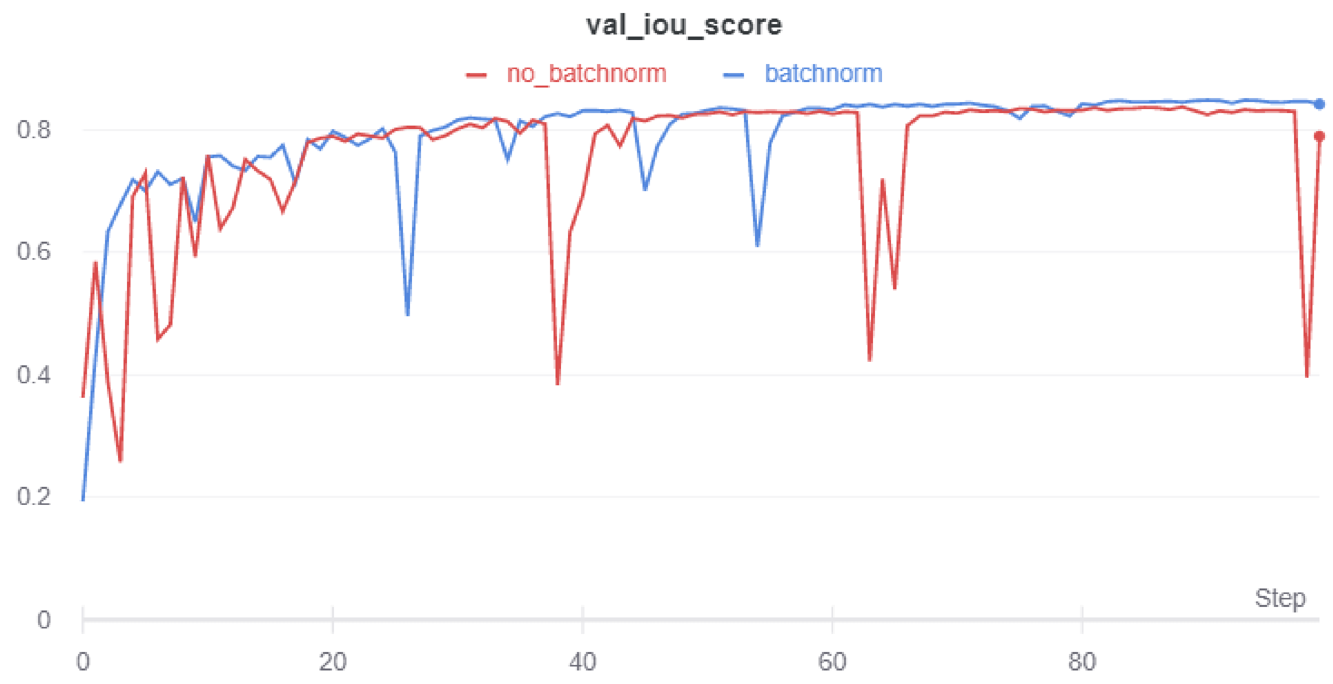
Batch normalisation is a technique that standardizes the inputs to a layer for each mini-batch. This has an effect of stabilising the learning process and reducing the number of epochs required to train deep networks. Batch normalisation scales the output of a layer, specifically by standardizing the activation of each input variable per mini-batch. Standardising the input means rescaling the input to have a mean of zero and a standard deviation of 1.

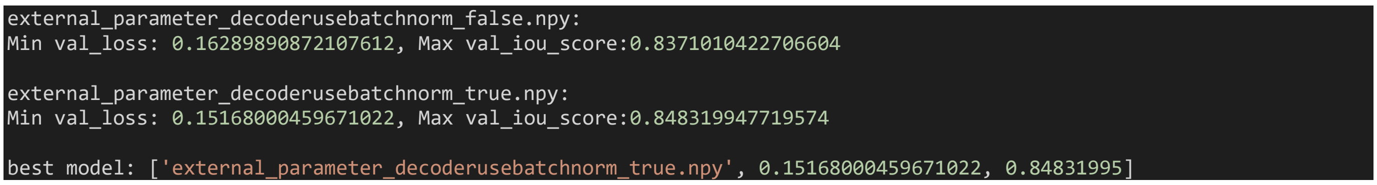
Standardising the activation of the prior layer means that the subsequent layer will be working with a largely consistent spread and distribution of inputs, and this has the effect of stabilising and speeding up the training process of deep neural networks. Additionally, it can have a regularising effect and improve the generalisation ability of the model. In our decoder, the exposed hyperparameter is to enable batch normalisation or to disable it, and we experiment to see if batch normalisation in our decoder portion of our Unet model is beneficial for the model.

Decoder Batch Normalization tested = [True, False]

Results







```
external_parameter_decoderusebatchnorm_false.npy:
Min val_loss: 0.16289890872107612, Max val_iou_score:0.8371010422706604

external_parameter_decoderusebatchnorm_true.npy:
Min val_loss: 0.15168000459671022, Max val_iou_score:0.848319947719574

best model: ['external_parameter_decoderusebatchnorm_true.npy', 0.15168000459671022, 0.84831995]
```

Fig 40: Results for batch normalisation against not using batch normalisation

Discussion

The performance of models with and without batchnorm are largely to our expectations. Batchnorm has been proven to be effective in deep neural networks, and our experiments confirm that. By standardising the inputs to a layer, the model performance also becomes more consistent, as can be seen by the batchnorm graph having less fluctuation in performance.

Decoder Drop Rate - Internal Hyperparameter

Dropout

Dropout is a regularisation method that approximates many neural networks with different architectures in parallel. During training, some number of layer outputs are randomly ignored. This makes the layer act differently during each epoch, as it has a different number of inputs and connectivity from the previous layer, and itself has certain outputs ignored, meaning it has a different combination of nodes active in every epoch.

Dropout makes the training process noisy, forcing nodes within a layer to probabilistically take on responsibility for the inputs. Because there is an infrequent combination of nodes and inputs each run, dropout reduces the co-dependency of nodes in a layer, and each node has to learn some weights that will influence the output, whereas without dropout, some nodes may constantly take most of the responsibility for the inputs, and in this way, dropout makes the model more robust.

Dropout simulates a sparse activation from a given layer and encourages the network to learn a sparse representation as a side-effect. During test time, we remove dropout, and the model benefits from leveraging an approximation of many neural networks, sort of like a less extensive version of ensemble modeling. Dropout is usually used to prevent overfitting, and we apply it with various drop rates throughout our model.

One thing to note is that dropout is theorised to clash with batch normalisation, because batch normalisation standardizes the input for each mini-batch, and dropout ignores some input each epoch, so the standardised input by batch normalisation if dropout is active changes drastically each epoch. Nonetheless, we experiment with dropout to see if this hypothesis applies to our specific use case, model, and dataset.

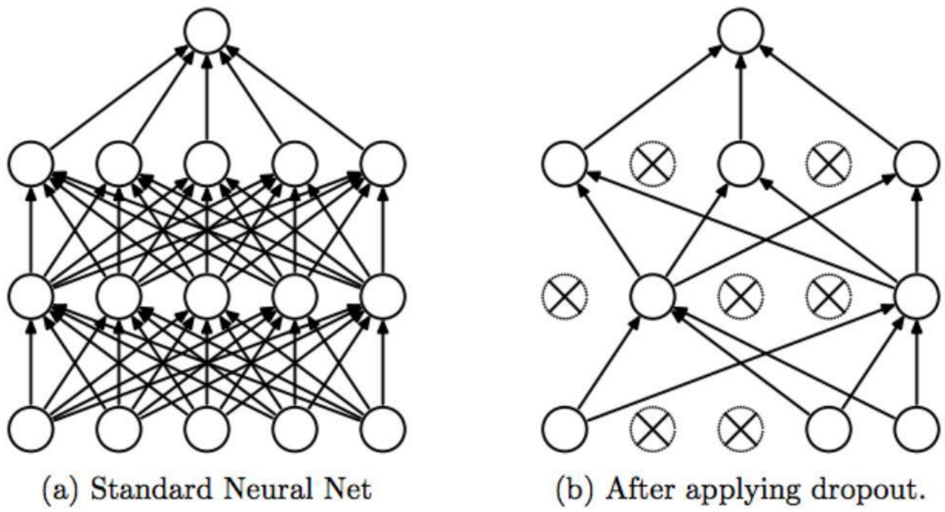


Fig 42 : Visualisation of dropout in a neural network

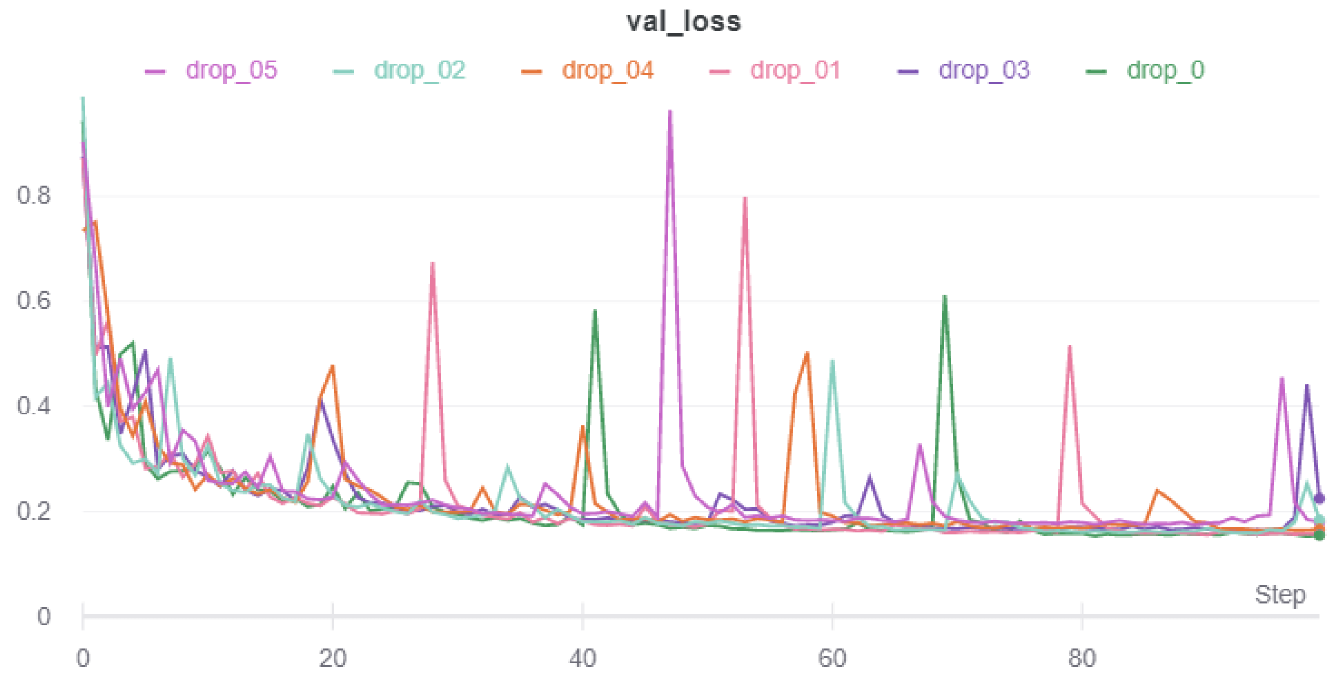
To enable dropout in our Unet decoder path, we modified the source code of **segmentation_models_dev** to propagate the dropout rate through the build_Unet function into DecoderUpsamplingX2Block, the Conv3x3BnReLU layer and finally into the Conv2dBn layer.

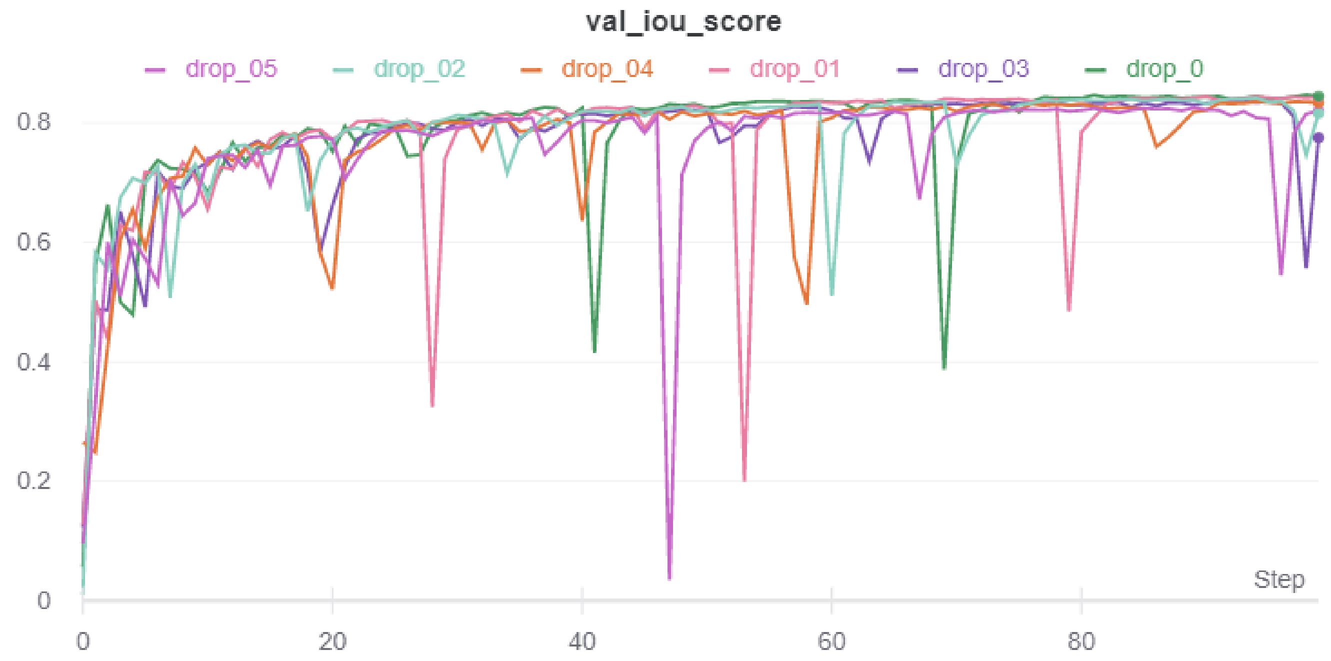
```
def Conv2dBn(
        filters, kernel_size,strides=(1, 1),padding='valid',data_format=None,dilation_rate=(1, 1),
        activation=None,kernel_initializer='glorot_uniform',bias_initializer='zeros',
        kernel_regularizer=None,bias_regularizer=None,activity_regularizer=None,kernel_constraint=None,
        bias_constraint=None,use_batchnorm=False,drop_rate=None,
        **kwargs
):
    """Extension of Conv2D layer with batchnorm"""

    def wrapper(input_tensor):

        x = layers.Conv2D(
            filters=filters,kernel_size=kernel_size,strides=strides,padding=padding,data_format=data_format,
            dilation_rate=dilation_rate,activation=None,use_bias=not (use_batchnorm),
            kernel_initializer=kernel_initializer,bias_initializer=bias_initializer,
            kernel_regularizer=kernel_regularizer,bias_regularizer=bias_regularizer,
            activity_regularizer=activity_regularizer,kernel_constraint=kernel_constraint,
            bias_constraint=bias_constraint,name=conv_name,
        )(input_tensor)

        if use_batchnorm:
            x = layers.BatchNormalization(axis=bn_axis, name=bn_name)(x)
        if drop_rate != None:
            x = layers.Dropout(rate=drop_rate, name=drop_name)(x)
        if activation:
            x = layers.Activation(activation, name=act_name)(x)
        return x

    return wrapper
```

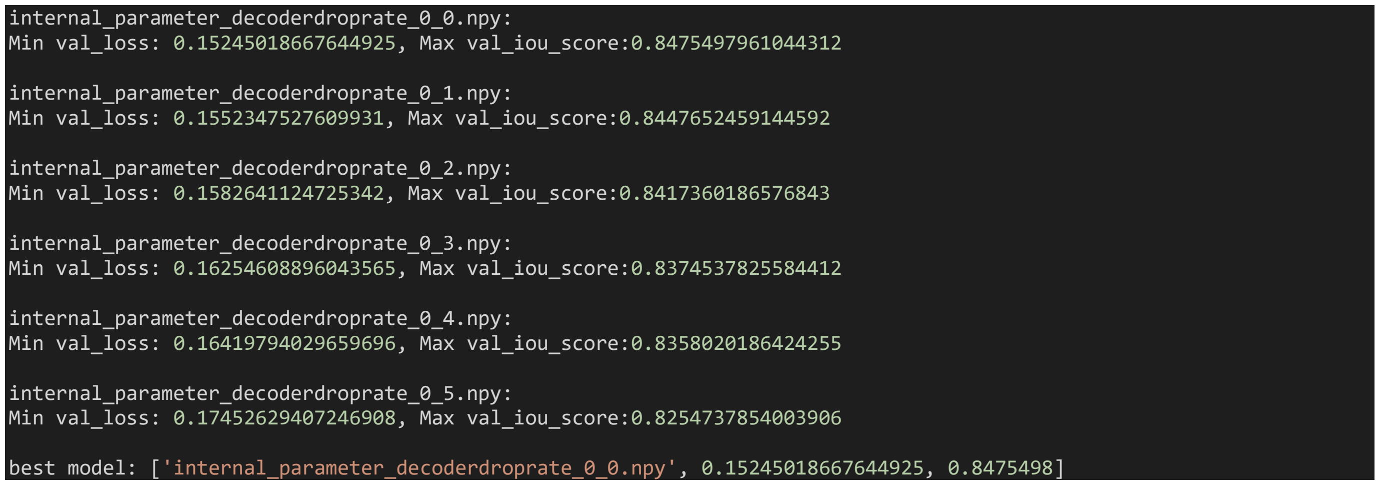
Code Block 41: Editing of source code to expose ability to include dropout in low-level layers to high-level API calls

Decoder drop rate tested = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5]

Results







```
internal_parameter_decoderdroprate_0_0.npy:
Min val_loss: 0.15245018667644925, Max val_iou_score:0.8475497961044312

internal_parameter_decoderdroprate_0_1.npy:
Min val_loss: 0.1552347527609931, Max val_iou_score:0.8447652459144592

internal_parameter_decoderdroprate_0_2.npy:
Min val_loss: 0.1582641124725342, Max val_iou_score:0.8417360186576843

internal_parameter_decoderdroprate_0_3.npy:
Min val_loss: 0.16254608896043565, Max val_iou_score:0.8374537825584412

internal_parameter_decoderdroprate_0_4.npy:
Min val_loss: 0.16419794029659696, Max val_iou_score:0.8358020186424255

internal_parameter_decoderdroprate_0_5.npy:
Min val_loss: 0.17452629407246908, Max val_iou_score:0.8254737854003906

best model: ['internal_parameter_decoderdroprate_0_0.npy', 0.15245018667644925, 0.8475498]
```

Fig 43: Results from various dropout drop rates

Discussion

Looking at the best performance of each model, we can see a clear trend that as dropout rate increases, the best performance of the model decreases. Various dropout rates cause different fluctuations in performance, but overall, we can see from the best performance of each model that dropout is detrimental to a model that already implements batch normalisation, as dropout causes the standardised input to fluctuate in every mini-batch.

Decoder Normalization - Internal Hyperparameter

We have already implemented batch normalization from our previous experiment. We will attempt to replace the batch normalization with group normalization.

Group Norm

The salient difference between group norm and batch norm is the input dimensions it normalizes over. Batch normalization normalizes all inputs across an entire channel C for an entire batch size N, whereas group norm normalizes inputs across several channels C as specified by the group size and ignores the batch size entirely. Typical applications of group norm are for cases where batch sizes are small, thus causing batch norm to perform poorly and train unstably. Our batch size of choice is 128, but whether this batch size is “large” or “small” can only be determined empirically.

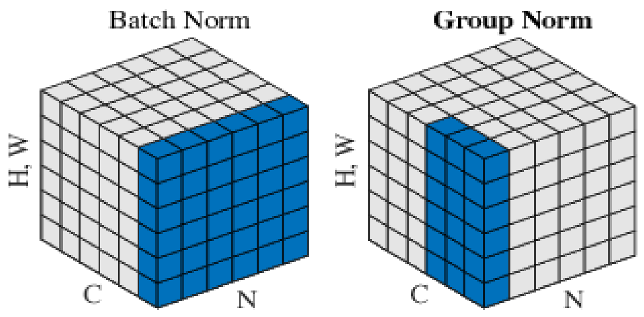


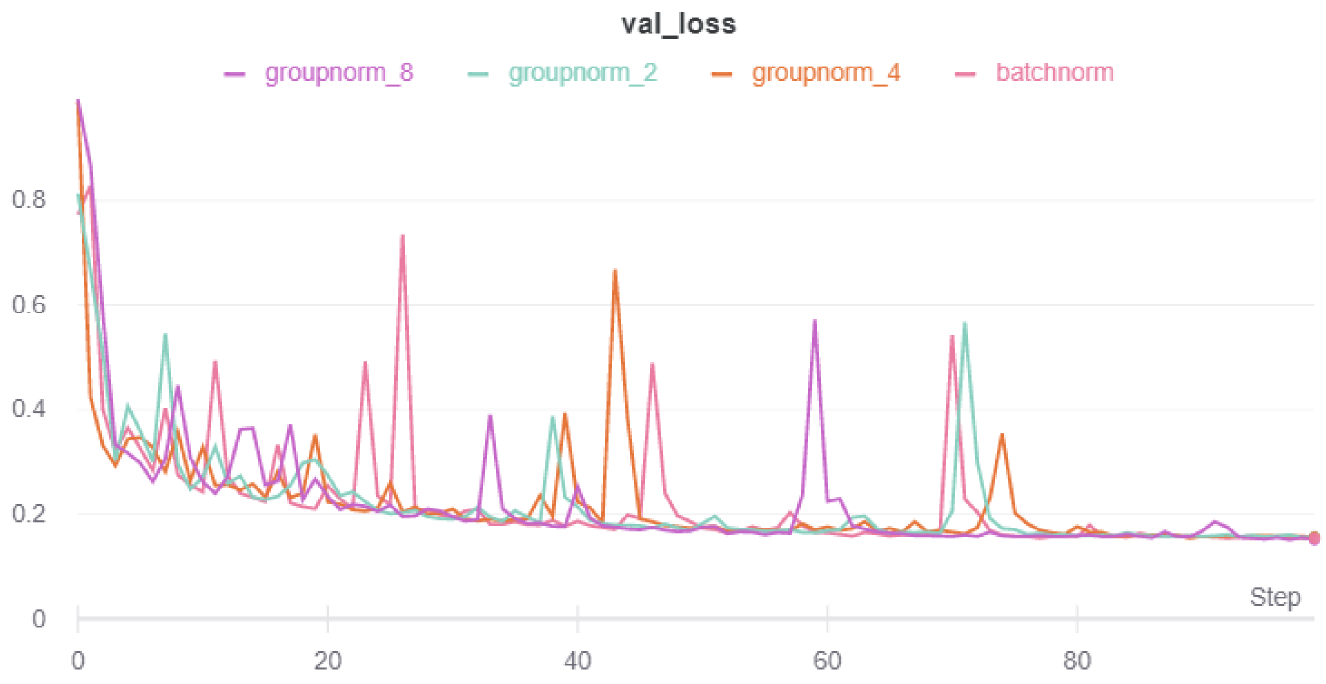
Fig 45 : Visualisation of batch norm vs group norm

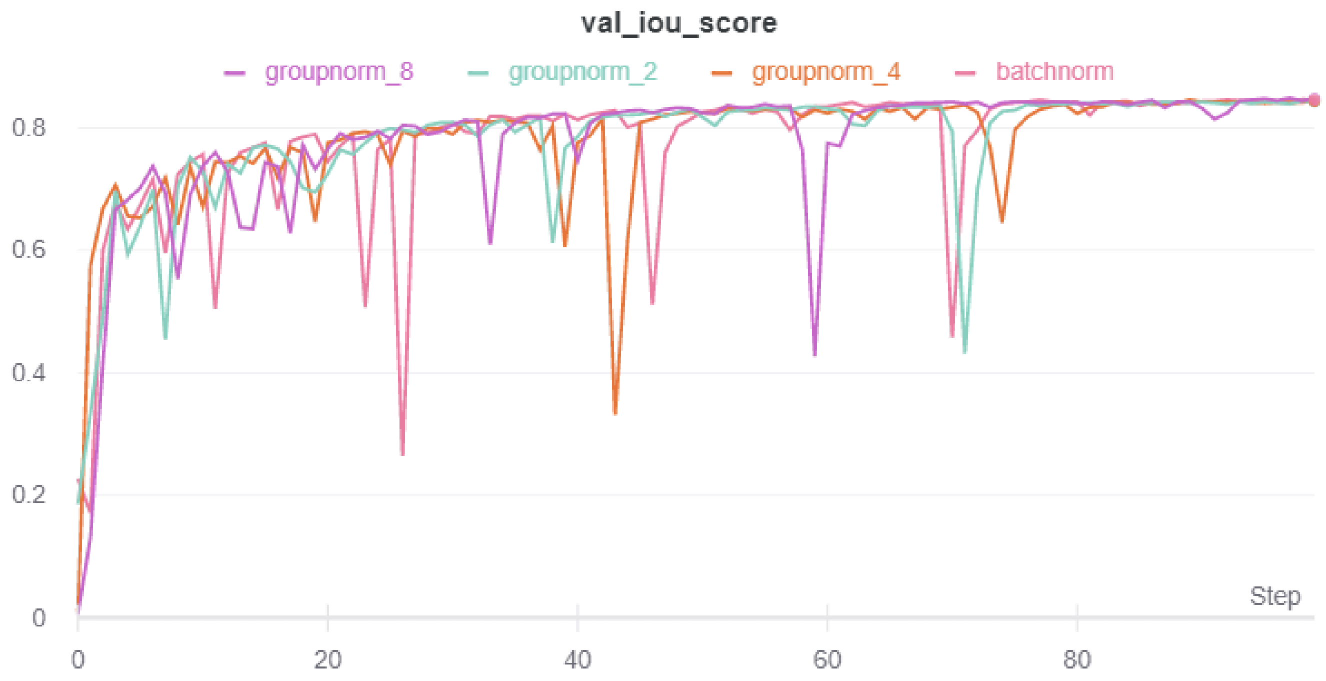
This requires us to change the Unet architecture’s decoder code in the **segmentation_models** package. We replicated the structure of the Conv2dBn layers and its calling functions with a Conv2dGn layer and allowed for changing of the group size for the model.

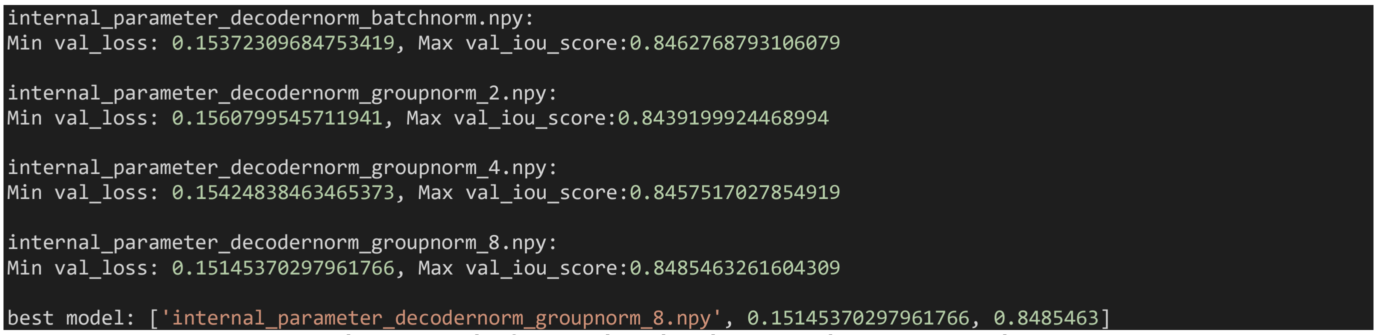
```
def Conv2dGn(
        filters,kernel_size,strides=(1, 1),padding='valid',data_format=None,dilation_rate=(1, 1),
        activation=None,kernel_initializer='glorot_uniform',bias_initializer='zeros',
        kernel_regularizer=None,bias_regularizer=None,activity_regularizer=None,kernel_constraint=None,
        bias_constraint=None,use_groupnorm=False,groupnorm_groups=None,drop_rate=None,**kwargs
):
    def wrapper(input_tensor):
        x = layers.Conv2D(
            filters=filters,kernel_size=kernel_size,strides=strides,padding=padding,data_format=data_format,
            dilation_rate=dilation_rate,activation=None,use_bias=not (use_groupnorm),
            kernel_initializer=kernel_initializer,bias_initializer=bias_initializer,
            kernel_regularizer=kernel_regularizer,bias_regularizer=bias_regularizer,
            activity_regularizer=activity_regularizer,kernel_constraint=kernel_constraint,
            bias_constraint=bias_constraint,name=conv_name,
        )(input_tensor)

        if groupnorm_groups == None and use_groupnorm:
            raise ValueError('Conflict in params detected. Check groupnorm_groups and use_groupnorm are coherent')
        if use_groupnorm and groupnorm_groups != None:
            x = tfa.layers.GroupNormalization(groups=groupnorm_groups, axis=gn_axis, name=gn_name)(x)
        if drop_rate != None:
            x = layers.Dropout(rate=drop_rate, name=drop_name)(x)
        if activation:
            x = layers.Activation(activation, name=act_name)(x)
        return x
    return wrapper
```
Code Block 44: Implemented low level Conv2d block to expose ability to use GroupNorm to high-level API calls

Decoder normalizations tested = [batchnorm, groupnorm(groups=2 | 4 | 8)]

Results







```
internal_parameter_decodernorm_batchnorm.npy:
Min val_loss: 0.15372309684753419, Max val_iou_score:0.8462768793106079

internal_parameter_decodernorm_groupnorm_2.npy:
Min val_loss: 0.1560799545711941, Max val_iou_score:0.8439199924468994

internal_parameter_decodernorm_groupnorm_4.npy:
Min val_loss: 0.15424838463465373, Max val_iou_score:0.8457517027854919

internal_parameter_decodernorm_groupnorm_8.npy:
Min val_loss: 0.15145370297961766, Max val_iou_score:0.8485463261604309

best model: ['internal_parameter_decodernorm_groupnorm_8.npy', 0.15145370297961766, 0.8485463]
```

Fig 46: Results from various batchnorm and groupnorm values

Discussion

All the models' best performance are within half a percentage point of each other. Looking at the validation loss and IOU over epochs graph, we also see that the performance of each model is largely similar. Hence, we can conclude that as batch size for batch normalisation is large enough, batch normalisation is effective and stable, and group normalisation does not provide any meaningful performance gains. However, we note that as the size of group normalisation increases, the performance of the model increases as well, and we can thus see that normalising over more channels helps to standardise the inputs better.

Residual Block Activations - Internal Hyperparameter

Residual Blocks

The heart of the ResNet architecture lies in residual blocks. A core issue with arbitrarily deep neural networks is the degradation problem, and the inability of these universal function approximators to learn the identity function. Hence, these residual blocks allow for an identity mapping of inputs to a future output and perform concatenation, effectively creating the skip architecture that prevents vanishing gradients. Conventionally, rectified linear units (ReLU) are used for the layer activation functions in residual blocks because of the broadly universal application and good performance.

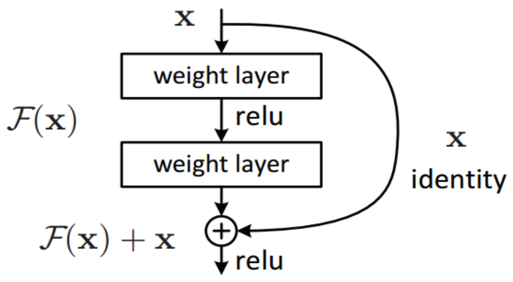
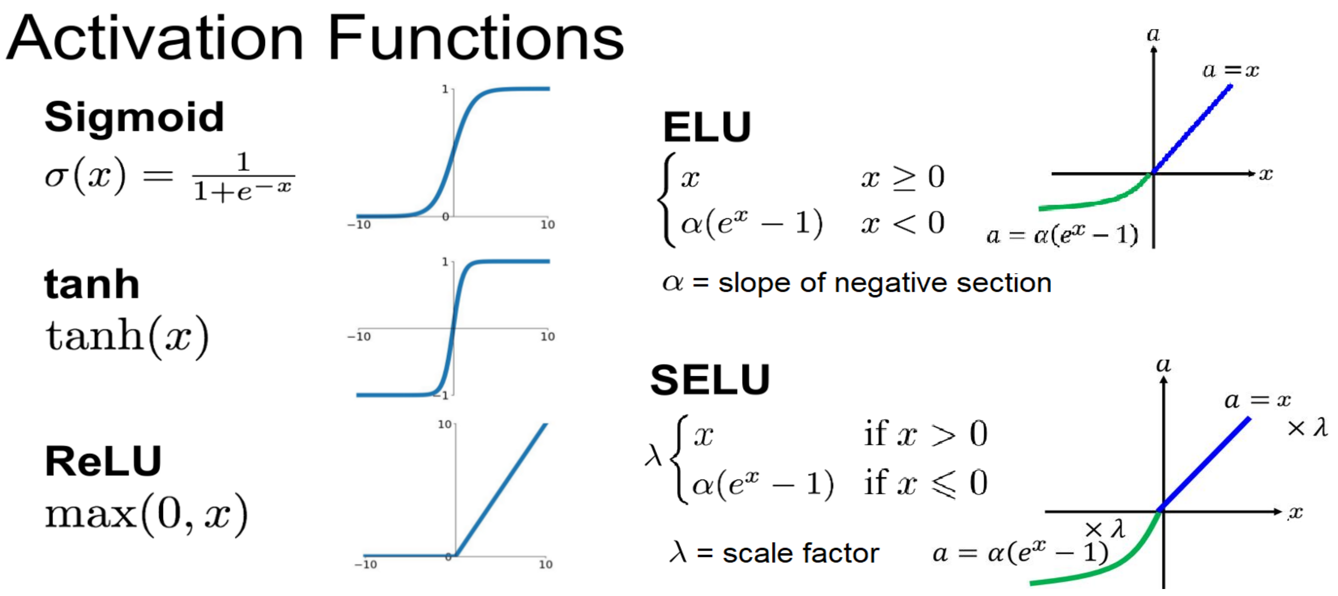


Fig 47 : Residual block in ResNets

Activations

However, there are numerous activation functions to select from, which perform well under different circumstances. Readily available activation functions from keras includes the following:

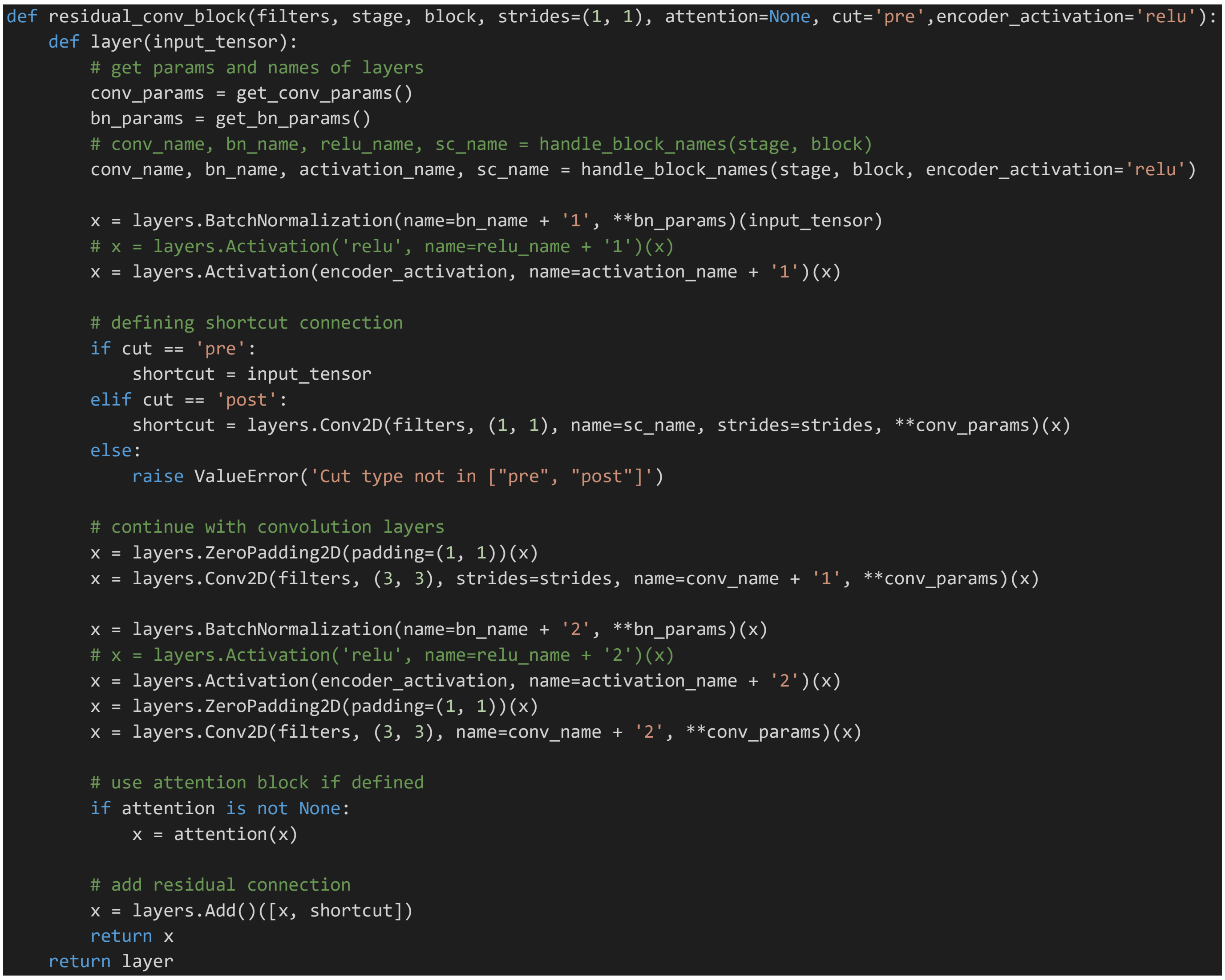


Activation Functions
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$
tanh
$\tanh(x)$
ReLU
$\max(0, x)$
ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$
α = slope of negative section
$a = x$
$a = \alpha(e^x - 1)$
SELU
$\lambda \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leqslant 0 \end{cases}$
λ = scale factor
$a = x$
$\times \lambda$
$a = \alpha(e^x - 1)$

Fig 49: Formula of various non-linear activations

The ResNet architecture however does not allow us to vary the layer activation function for residual blocks. Therefore, we must change the source code again and add a **resnet18_modified** to the **model_factory** pipeline of the **classification_models_dev** package to accept different activation functions for residual block layers.

Refer to **classification_models_dev/classification_models_dev/models/resnet18_modified.py** for our modifications.



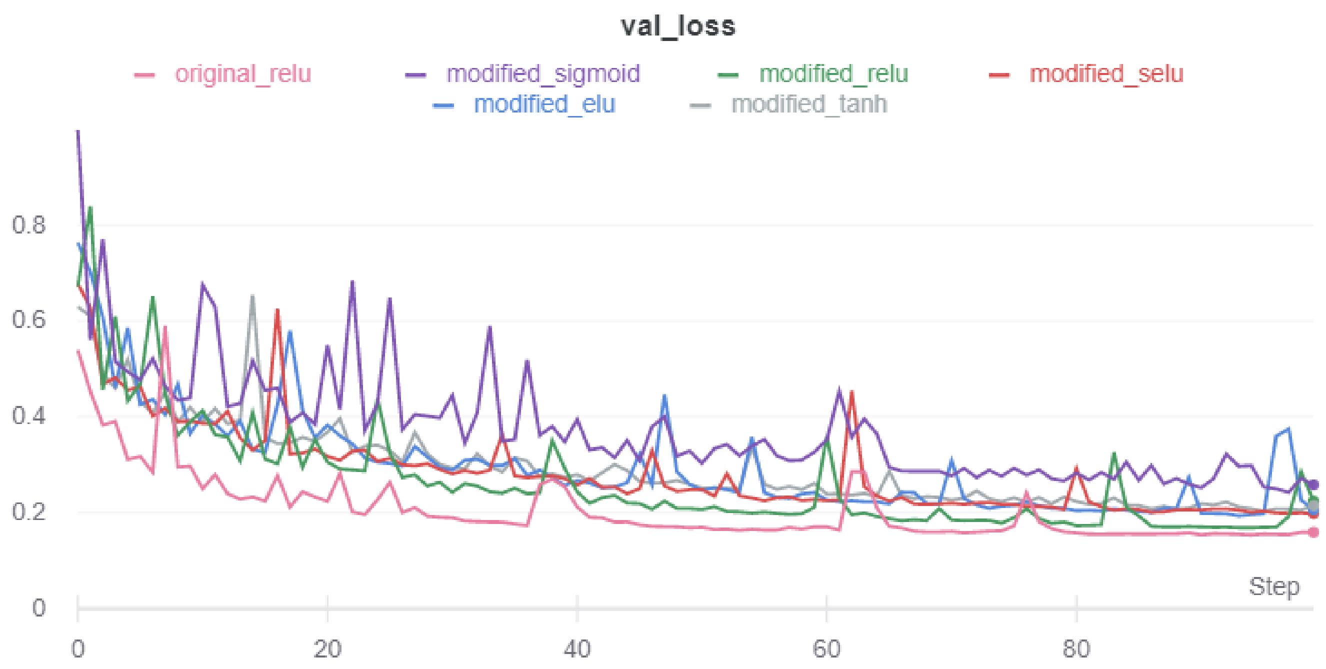
```
def residual_conv_block(filters, stage, block, strides=(1, 1), attention=None, cut='pre',encoder_activation='relu'):
    def layer(input_tensor):
        # get params and names of layers
        conv_params = get_conv_params()
        bn_params = get_bn_params()
        # conv_name, bn_name, relu_name, sc_name = handle_block_names(stage, block)
        conv_name, bn_name, activation_name, sc_name = handle_block_names(stage, block, encoder_activation='relu')

        x = layers.BatchNormalization(name=bn_name + '1', **bn_params)(input_tensor)
        # x = layers.Activation('relu', name=relu_name + '1')(x)
        x = layers.Activation(encoder_activation, name=activation_name + '1')(x)

        # defining shortcut connection
        if cut == 'pre':
            shortcut = input_tensor
        elif cut == 'post':
            shortcut = layers.Conv2D(filters, (1, 1), name=sc_name, strides=strides, **conv_params)(x)
        else:
            raise ValueError('Cut type not in ["pre", "post"]')

        # continue with convolution layers
        x = layers.ZeroPadding2D(padding=(1, 1))(x)
        x = layers.Conv2D(filters, (3, 3), strides=strides, name=conv_name + '1', **conv_params)(x)

        x = layers.BatchNormalization(name=bn_name + '2', **bn_params)(x)
        # x = layers.Activation('relu', name=relu_name + '2')(x)
        x = layers.Activation(encoder_activation, name=activation_name + '2')(x)
        x = layers.ZeroPadding2D(padding=(1, 1))(x)
        x = layers.Conv2D(filters, (3, 3), name=conv_name + '2', **conv_params)(x)

        # use attention block if defined
        if attention is not None:
            x = attention(x)

        # add residual connection
        x = layers.Add()([x, shortcut])
        return x
    return layer
```

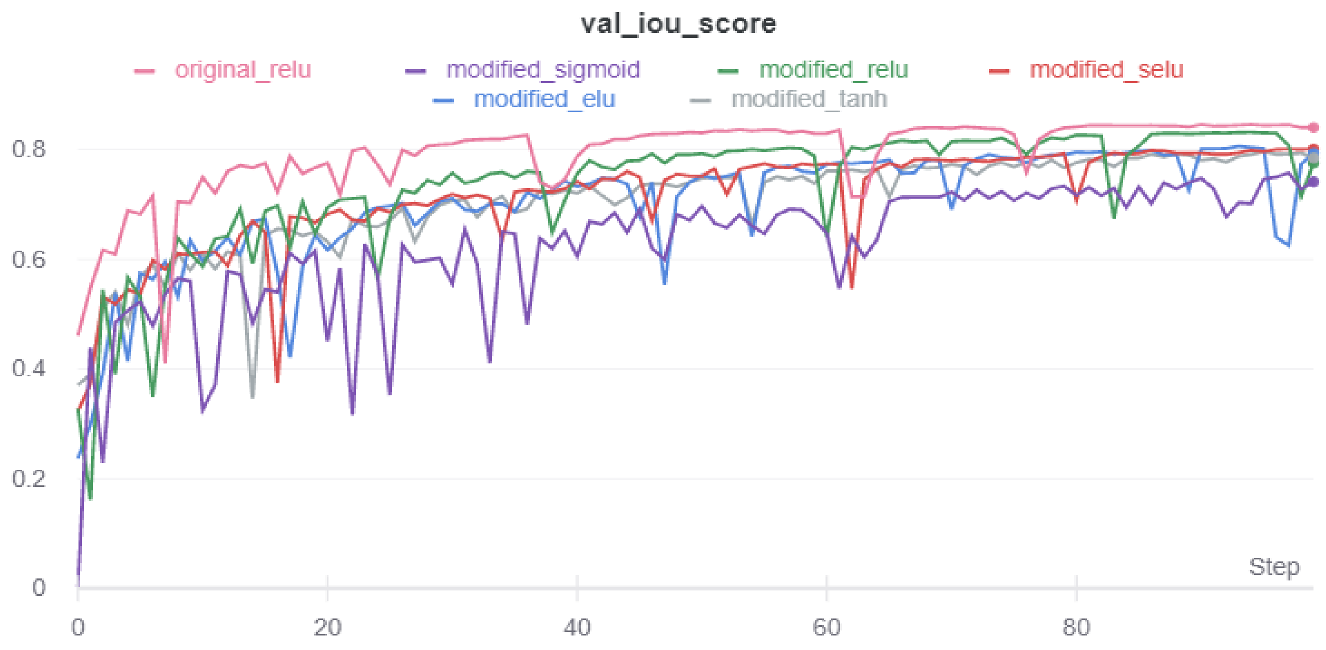
Code Block 48: Modifying resnet backbone to allow for changing of activation functions

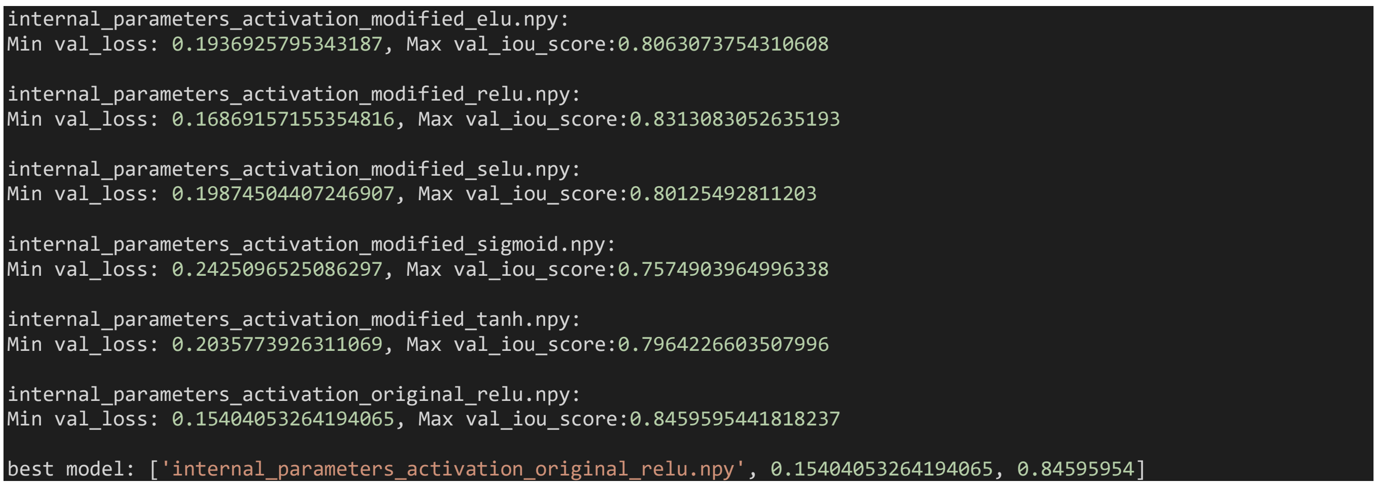
While we experiment with activation functions, it is likely that ReLU will perform the best. We also note that we cannot load weights from imagenet into our model for the other activation functions that we use since the weights do not correspond to ReLU. For comparison, “modified_relu” refers to using the ReLU activation without pretrained weights, which ends up still outperforming all other activation functions without pretrained weights.

Residual block activations tested = [elu, relu, selu, sigmoid, tanh, relu (original, no pretrained weights)]

Results







```
internal_parameters_activation_modified_elu.npy:
Min val_loss: 0.1936925795343187, Max val_iou_score:0.8063073754310608

internal_parameters_activation_modified_relu.npy:
Min val_loss: 0.16869157155354816, Max val_iou_score:0.8313083052635193

internal_parameters_activation_modified_selu.npy:
Min val_loss: 0.19874504407246907, Max val_iou_score:0.80125492811203

internal_parameters_activation_modified_sigmoid.npy:
Min val_loss: 0.2425096525086297, Max val_iou_score:0.7574903964996338

internal_parameters_activation_modified_tanh.npy:
Min val_loss: 0.2035773926311069, Max val_iou_score:0.7964226603507996

internal_parameters_activation_original_relu.npy:
Min val_loss: 0.15404053264194065, Max val_iou_score:0.8459595441818237

best model: ['internal_parameters_activation_original_relu.npy', 0.15404053264194065, 0.84595954]
```

Fig 50: Results for various non-linear activations

Discussion

In both validation loss and IOU, original ReLU performed better than other non-linear activations. It also had the most stable performances out of all the non-linear activations.

We can see that ReLU activations perform the best, both with and without imagenet pretrained weights. The other activation functions perform similarly to each other, except for Sigmoid, which is clearly worse than every other activation function for our purpose.

The sigmoid activation function had been the non-linear activation function of choice for many years due to its intuitive interpretation as probability. However, the sigmoid function suffered from the vanishing gradient problem, observable from the curve above. When the synaptic input is large, the sigmoid function takes on values close to 0 and 1. In these regions, the function plateaus, and gradients in these regions are close to 0. This presents a problem for learning of neural networks, which relies on gradient updates to optimize parameters towards a minimum loss. The vanishing gradients then lead to poor learning for the neural networks.

As for the ReLU activation function, it avoids the issue of vanishing gradients due to large synaptic inputs, as it always maps the value of the input to itself, while being a nonlinear function as all negative inputs are suppressed to 0. In fact, the suppression of negative synaptic inputs leads to sparsity in the neural networks, where many activations are equal to 0. This sparsity contributes to better space and time complexity, and reduces computational overhead when training the neural network, as it does not involve the computationally expensive exponential function. Moreover, this sparsity has been observed empirically to greatly aid neural network learning.

While the above-mentioned properties of ReLU leads to the desirable sparse network, we may encounter dead ReLUs, when the synaptic input is largely negative, and our corresponding outputs and gradients are then zero, leading to poor training. Newer activation functions like ELU and SELU have been introduced to prevent this problem by having a non-zero output in the negative input regions. Ultimately, in this problem setting, we found experimentally that ReLU still outperformed these activation functions.

Data Augmentation

```
def load_dataset(filenames, augment=False):
    ignore_order = tf.data.Options()
    ignore_order.experimental_deterministic = False
    dataset = tf.data.TFRecordDataset(
        filenames
    )
    dataset = dataset.with_options(
        ignore_order
    )
    dataset = dataset.map(
        decode_record, num_parallel_calls=tf.data.experimental.AUTOTUNE
    )

    if augment:
        dataset = dataset.map(
            data_augment, num_parallel_calls=tf.data.experimental.AUTOTUNE
        )

    return dataset
```

Code Block 51: Data augmentation as part of our tf.data.Dataset pipeline

Increasing our dataset helps our model be able to generalise better, but this is contingent on our augmented data. We should augment the data to simulate information loss or variation in data that is within the variation of the dataset we collect. As we are dealing with satellite images, we need to model the variation in quality and direction these images could have been collected. We chose to experiment with augmentations that we knew could affect RGB images taken as satellite images. We included rotation of the image, flipping left-to-right and upside-down as the satellite could pass over the area imaged in many different orientations, and we sought to mimic different conditions the images could have been taken. Due to the weather and time the images were taken, the coloration of the images could be varying as well. Lastly, we tried gaussian noise to add resilience to the model. We added the data augmentations as a pipeline in the data preparation for each batch with random probability of augmentation, and in this way, we 'created' additional training samples without increasing computation time as we were training on the same amount of samples per epoch.

Then, we ran each augmentation individually, to see which of them improved the accuracy of the model, and from there, we created a data augmentation pipeline to include those augmentations that were beneficial to the generalisation capability of our model.

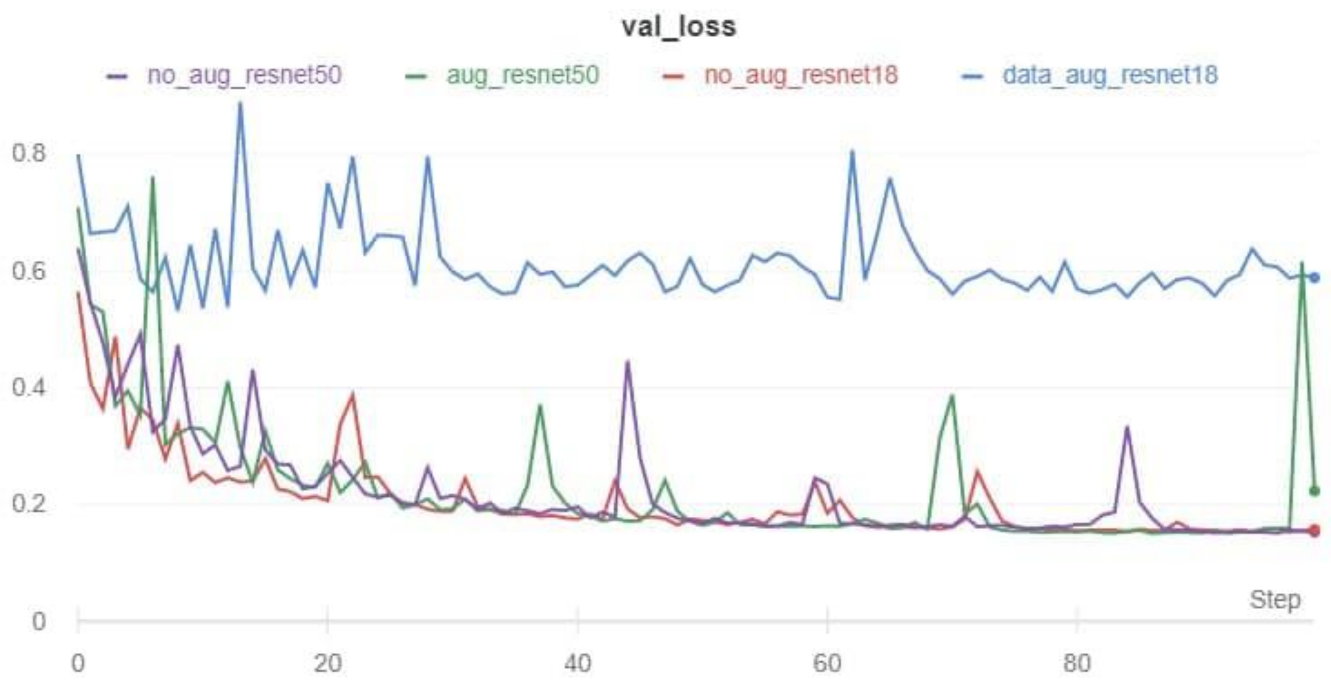
Not all data augmentations are helpful or meaningful. GIven that we are dealing with satellite images, we can expect most of them to look very similar because satellite images are typically taken by very few satellites, which generally take images and process them in very similar fashion. We do not anticipate a lot of perspective shifts, warping, or even noise for that matter because satellite images are typically of high quality as they come from highly expensive equipment.

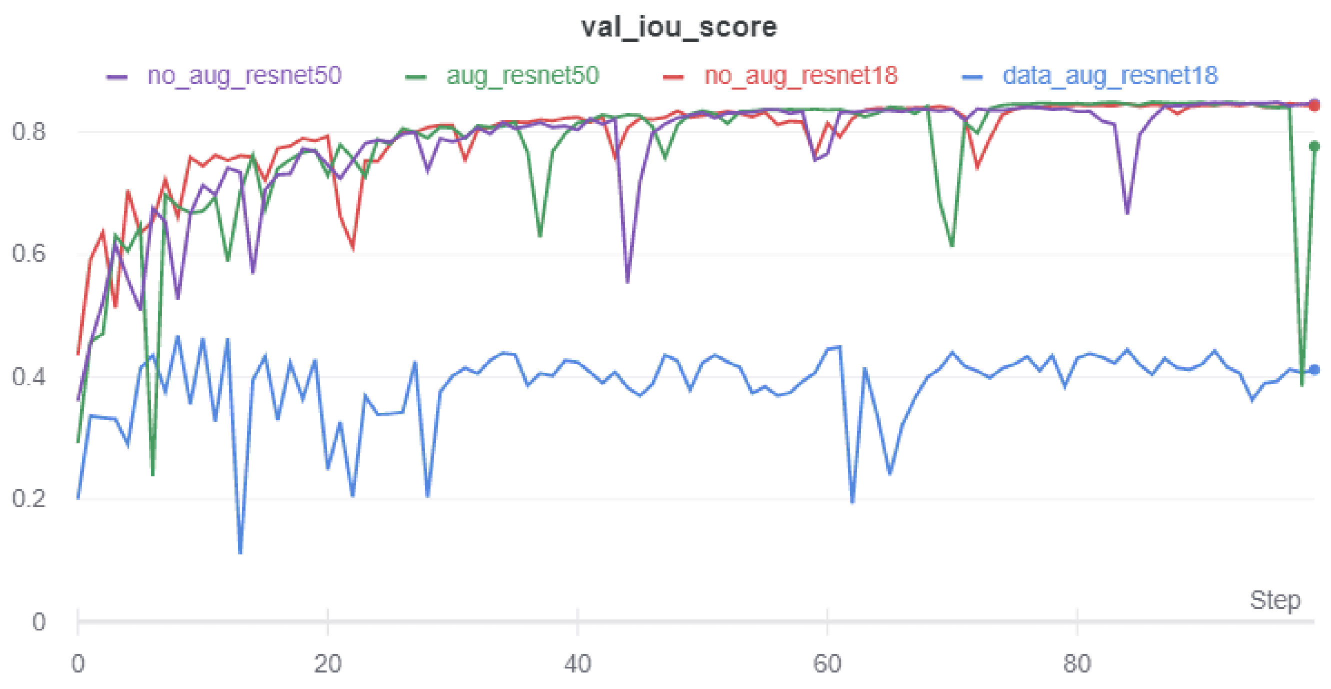
We tested each augmentation individually and chose augmentations and probability values based on how much they contributed to our model's performance.

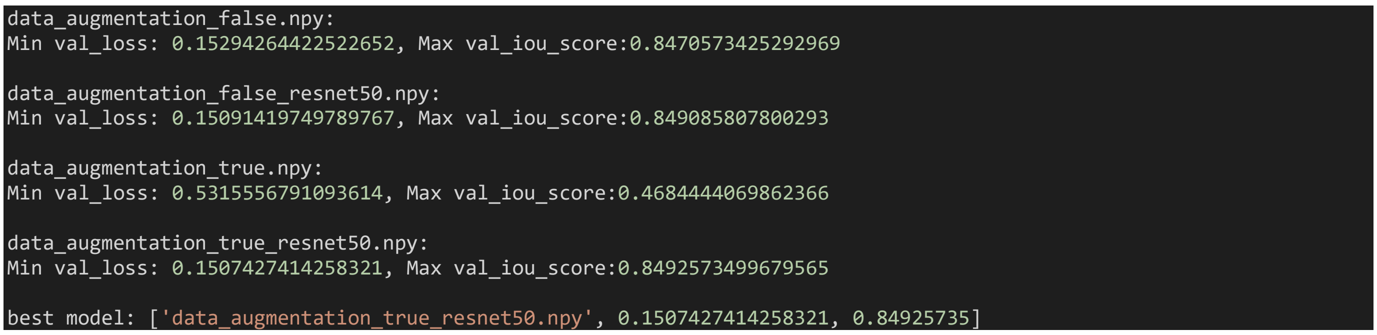
Our final data augmentation probability parameters selected are:

rot90_prob=0.2, fliplr_prob=0.2, gamma_prob=0.2.

Results







```
data_augmentation_false.npy:
Min val_loss: 0.15294264422522652, Max val_iou_score:0.8470573425292969

data_augmentation_false_resnet50.npy:
Min val_loss: 0.15091419749789767, Max val_iou_score:0.849085807800293

data_augmentation_true.npy:
Min val_loss: 0.5315556791093614, Max val_iou_score:0.4684444069862366

data_augmentation_true_resnet50.npy:
Min val_loss: 0.1507427414258321, Max val_iou_score:0.8492573499679565

best model: ['data_augmentation_true_resnet50.npy', 0.1507427414258321, 0.84925735]
```

Fig 52: Results for data augmentation

Discussion

We observed that data augmentation performed exceptionally poorly for the ResNet18 model. This was surprising because data augmentation is usually helpful for most situations. We hypothesized that the possibility lies with the number of layers in our architecture - ResNet18 only has 18 layers, which might be insufficient to learn all the characteristics and features of our augmented dataset. Deeper networks might be able to learn more effectively and utilise the extra training samples that are provided from our data augmentation.

We ran the model again utilising ResNet50 as a backbone to investigate this hypothesis and found that indeed, data augmentation will help our model since ResNet50 with data augmentation is now our best performing model, albeit by a very negligible amount (only 0.00025 more val_iou_score than ResNet50 unaugmented). However, this demonstrates that deeper models will be able to make use of an enhanced dataset to learn more features, while shallower models like ResNet18 might be able to perform well with simpler datasets, but not be able to adapt as effectively.

However, for consistency across all our experiments, we will stick with the ResNet18 model without data augmentation to be our best model.

Findings

Summary of best models and hyperparameters

S/N	Experiment	Best Model	val _loss	val_iou _score	test _loss	test_iou _score
1	Architecture Trial	resnet18	0.0143	0.7700	0.0125	0.7833
2	Loss Function	JaccardLoss	0.1531	0.8468	0.1318	0.8681
3	Optimization Algorithm	Adam	0.1550	0.8449	0.1391	0.8608
4	Learning Rate	10e^{-4}	0.1563	0.8436	0.1555	0.8444
5	Decoder Block Type	Upsampling	0.1522	0.8477	0.1396	0.8603
6	Decoder Batch Normalization	True	0.1516	0.8483	0.1530	0.8469
7	Decoder Drop Rate	0.0	0.1524	0.8475	0.1377	0.8622
8	Decoder Normalization	Group Normalization Group Size = 8	0.1514	0.8485	0.1435	0.8564
9	Residual Block Activations	Original ReLU	0.1540	0.8459	0.1542	0.8457
10	Data Augmentation	False	0.1529	0.8470	0.1296	0.8703
**	Data Augmentation + ResNet50 → not officially part of our experiments	True, ResNet50	0.150	0.8492	0.1517	0.8482

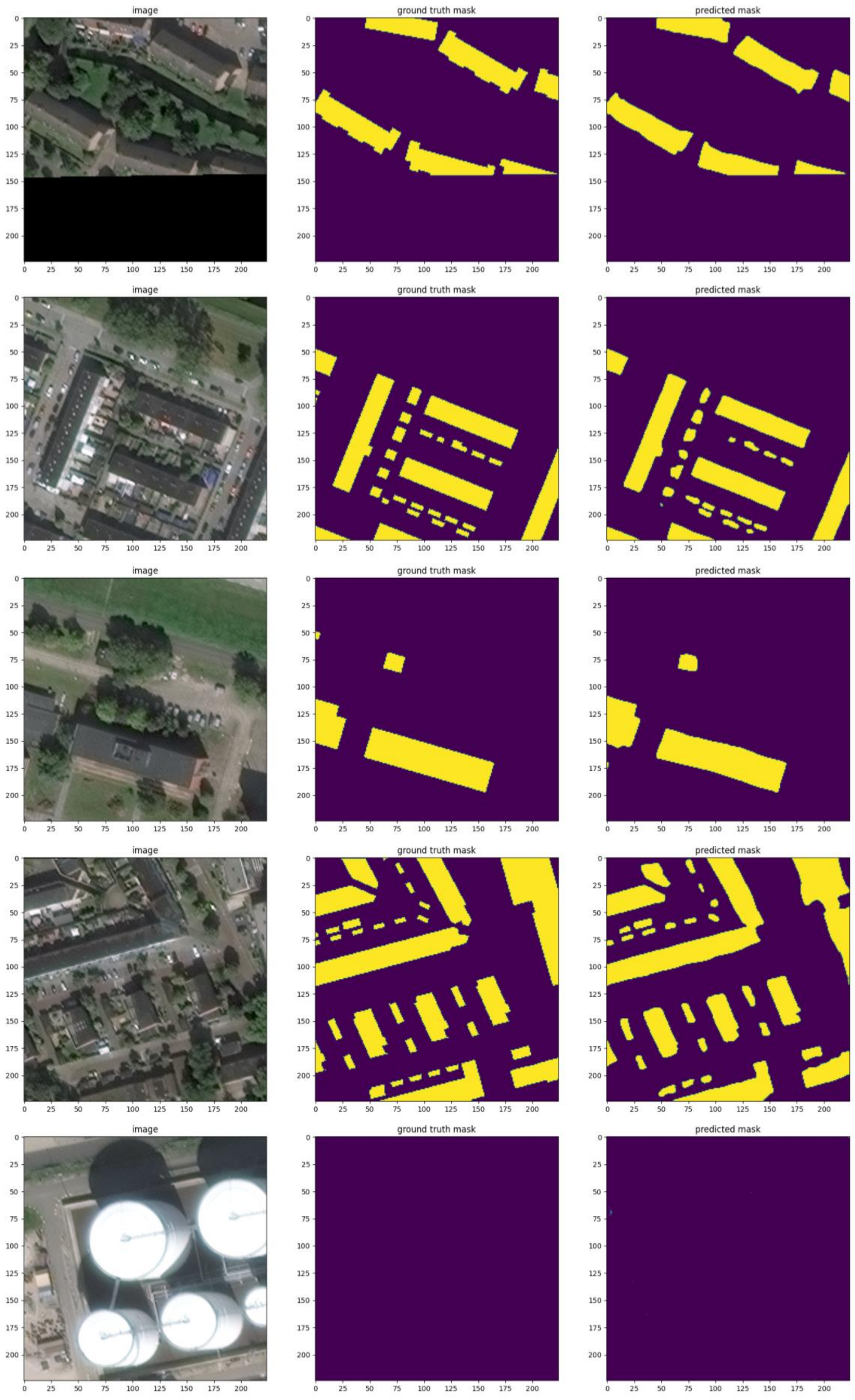
We observe that the most impactful change was from the loss function (yellow). Beyond that, the other experiments mostly produced very incremental benefits. However, we did end up with a final model that has the highest test_iou_score on the test set by our very last experiment (green).

Predictions

We have plotted out some of the predictions (128 of them) that we have obtained from the best model (data_augmentation_false.py) and saved them to ***results/sample_figs***. Here are some of the predictions that we have. We also categorise them according to the certain characteristics of the prediction.

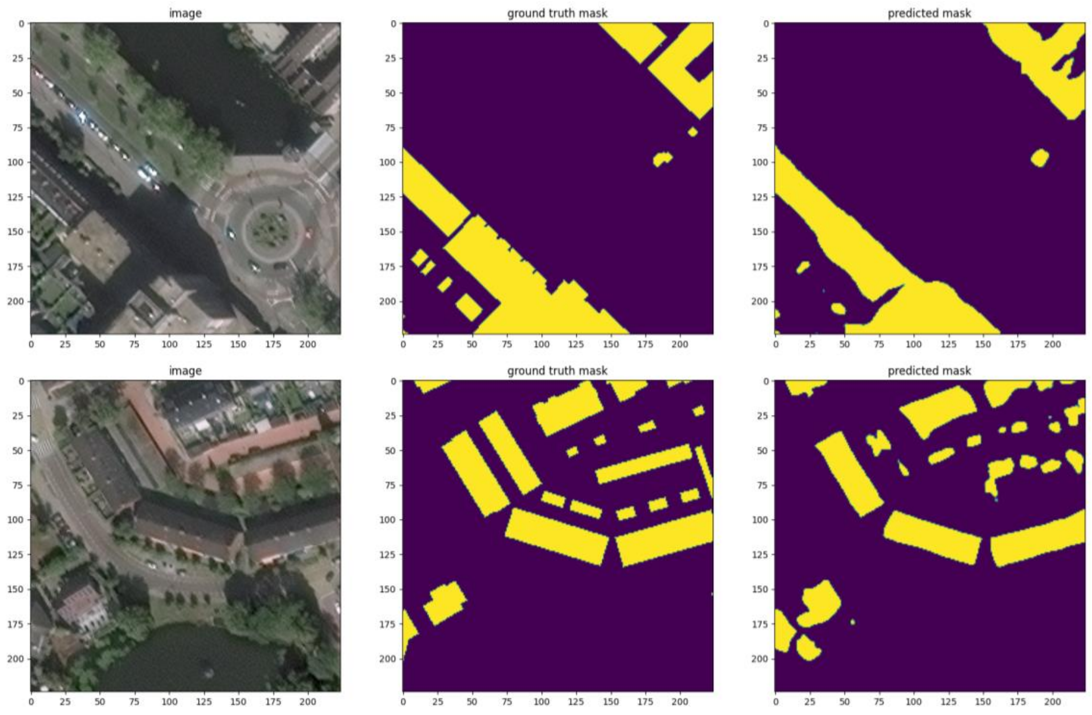
Good Predictions

These are predictions that match the ground truth mask quite well!



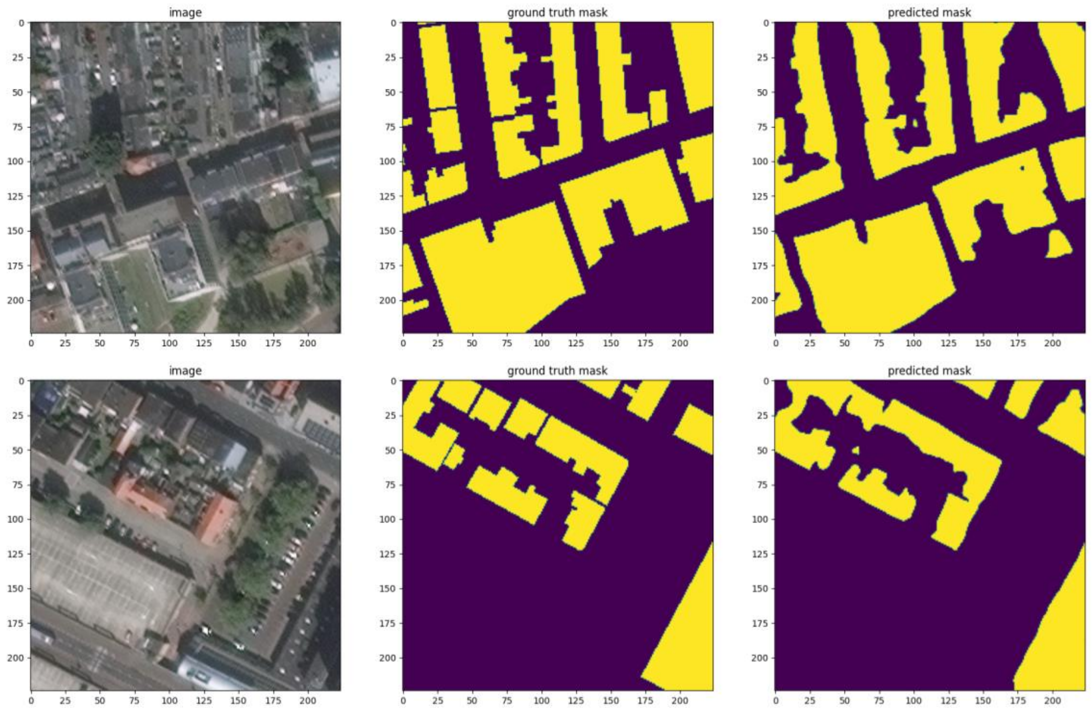
Unable to differentiate/identify small buildings

Sometimes, small buildings are simply clumped together with the larger buildings or missed entirely.



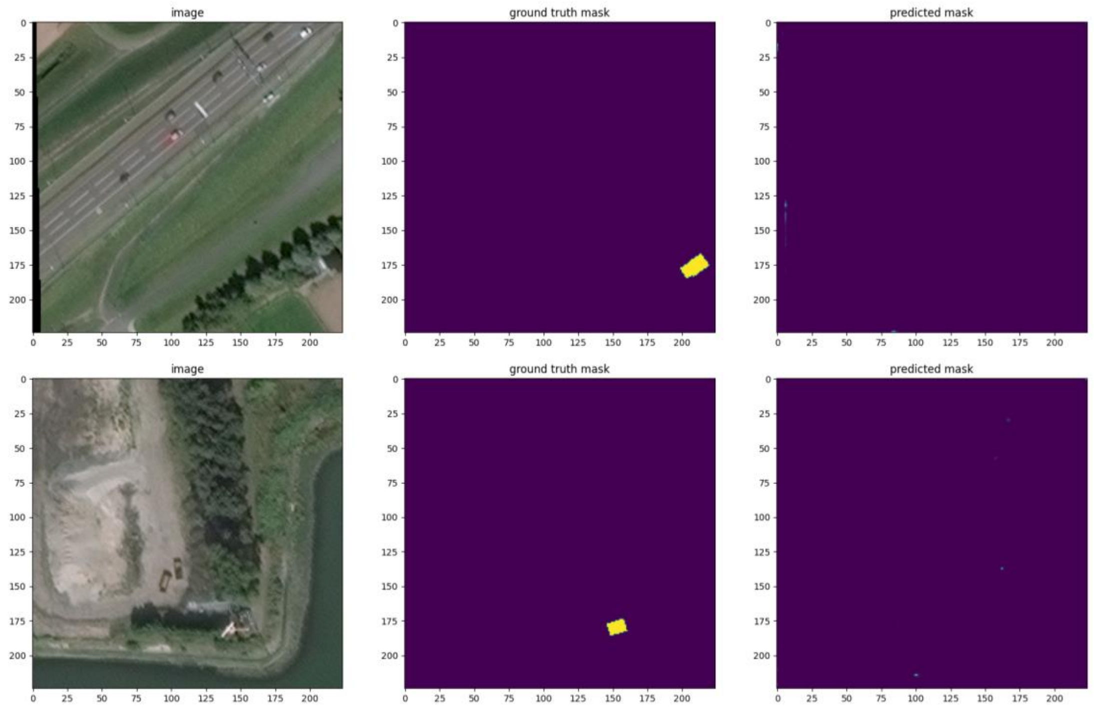
Poor building contours

This is marked by the inability of the model to draw straight lines or smooth contours for building edges, which is the natural geometric tendency for buildings.



Missed predictions

These usually occur for very small and isolated buildings in the entire image



Further Improvements

Given more resources and time, we would like to perform experiments on larger models, such as Resnet-152, as well as Efficientnet models. Efficientnet models increase the dimensions of our model and were too big to fit given our memory constraints. The top performing models for Spacenet 6 all used ensemble models. Because we were limited by resources and time, we could not implement ensemble models, or utilise the SAR data effectively. Moving forward, given increased resources, we would like to take advantage of the data-rich SAR images to train our model, giving more value to first responders to humanitarian disasters on the ground.

For mask generation, we are currently using binary segmentation, but we can refine it to include building edge masks as well as building interior masks, to improve our detection of buildings. This might help in solving the fine-grain predictions involving small buildings, because edge detection might prevent smaller buildings from being absorbed by the larger ones.

Lastly, we can consider upgrading the experiment from semantic segmentation (building or not) to instance segmentation (building A, building B…). This will become a harder segmentation problem. However, it has important applications in the real world that simple semantic segmentation cannot answer. Semantic segmentation may be useful for broader applications like identifying building footprints in a disaster area because all buildings are targets of interest. However in the military context, identifying specific buildings among a cluster (e.g. headquarters vs cookhouse) will be essential towards their operations, and require more detailed analysis of the satellite imagery to make intelligent inference about the instance of a building, and its type.

Conclusion

Our project aim was to explore various model architectures, exposed hyperparameters and internal hyperparameters to create a model that gives us the best prediction in the Spacenet 6 challenge of building segmentation. We utilised Unet architecture and found that Resnet-18 and Upsampling2D block had the best performance. Over various experiments, we found that JaccardLoss was the loss function that improved our evaluation metric the most, Adam optimizer with learning rate $10e^{-4}$, and activation function of ReLU improved performance. Batch normalisation with GroupNorm (size=8) helped our model, but Dropout clashed with batch normalisation and so was left out. Data augmentation improved our deeper models but did not help our Resnet-18 model. All in all, we are proud of our ability to find 87 percent of buildings in our segmentation model, which we are sure would be useful for the practical reasons mentioned above.

<<< END >>>