Road Segmentation with ШЩ-Net: a Deeper Version of UNet

Ünsal Öztürk
EPFL - 320826
unsal.ozturk@epfl.ch

Mert Soydinç
EPFL - 321131
mert.soydinc@epfl.ch

Utku Görkem Ertürk
EPFL - 321977
utku.erturk@epfl.ch

Abstract—We train several classifiers to segment roads from aerial images and compare their performances on the aerial images dataset. As a baseline, we train a logistic regressor operating on 16x16x3 mini-patches of the image. As for more advanced models, we start by training a CNN operating on 64x64x3 patches with pixel-wise output. We further extend our classifier toolkit by exploring more recent deconvolutional architectures, based on UNets. We establish the 'vanilla' UNet as a baseline deconvolutional model and expand upon this model by increasing the depth of this neural network. We propose two other architectures, which we call Ш-Net and ШЩ-Net, that resemble UNet architecture in terms of skip connections, encoding and decoding. We perform experiments on multiple augmentations of the initial dataset with all these classifiers, and different loss functions. We conclude that a ШЩ-Net with cross-entropy loss achieves the best results: 95.0% accuracy and has an F1 score of 0.909 on the entire test set.

I. Introduction

We are given a training dataset of 100 color images with sizes 400x400, along with its road labels. The prediction task in question involves segmenting 50 color aerial images of size 608x608 such that roads are assigned a pixel-wise label of 1 and the rest of the image is assigned a label of 0. The submission system for the evaluation of our models quantizes the output by patches of size 16x16, and we therefore quantize the output of our models before submission.

We attempt to solve this classification problem using five models: logistic regression, a basic CNN, a UNet, a UNet with an extra decoder with skip connections, which we call Ш-Net, and a Ш-Net feeding into a UNet, which we call ШЩ-Net. We lay out the architectures for these models, provide experimental results using multiple data augmentations and loss functions.

We implement our models, augmentation, and evaluation on PyTorch [1], Numpy [2], scikit-image [3] and SciPy [4]. We do not use any external datasets to stay faithful to the original challenge.

II. Models

A. Logistic Regression

We use logistic regression as a baseline model, which is distributed along with the dataset. The model crops up the image into patches of size 16x16 and trains a logistic regressor over the pixels in this patch, the output of which for the whole patch is 0 or 1 depending on the output of the regressor.

B. Basic CNN

The basic sequential CNN model aims to achieve better classification accuracy compared to logistic regression. The model comprises of four convolutional layers with ReLU activations and dropouts, followed by 2x2 max pools. The convolutional part of the network then feeds into a dense part with an input layer size of 1024, followed by a hidden layer of size 128, which then feeds into an output of 2 neurons (cross-entropy) or 1 neuron (MSE), depending on the experiment.

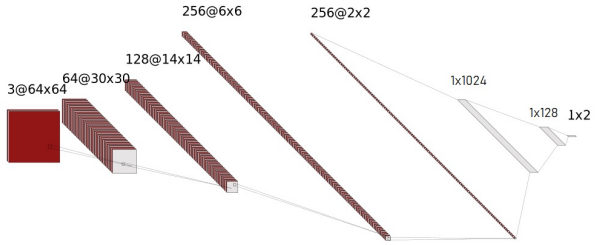


Figure 1: Architecture of Basic CNN

C. Deconvolutional Neural Networks

1) UNet: We implement a very similar version of UNet as proposed in [5]. This network is comprised of an encoding part and a decoding part. Each node in the encoding part performs two convolutions and performs a max pooling to double the number of channels and halve the dimensions of the image. We use Leaky ReLU instead of regular ReLU to prevent gradients from becoming zero. The decoding part of the network halves the number of channels and doubles the dimensions of the image per node. The encoder is connected to the decoder with skip connections where image dimensions agree. We assume the input to our implementation to have dimensions divisible by 16. The output size is the same as the input size.

2) Ш-Net: This model is an extension of UNet. We obtain this model by placing convolutional layers in the middle of the skip connections and connecting these new convolutional layers to one another with upsampling layers. This neural network is also fully

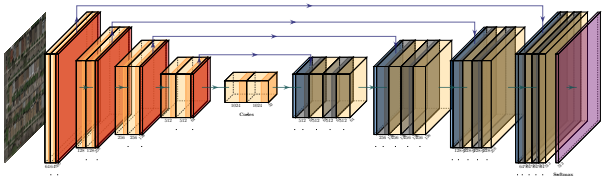


Figure 2: Architecture of UNet

convolutional and supports images of all sizes, given that the dimensions of the image are divisible by 16. Non-UNet parts in Figure 3 depict the architecture of the network (higher resolution available in our GitHub repository).

3) ШЦ-Net: This model is an extension of Ш-Net. Essentially, we connect the output of a Ш-Net, beside its last activation, to the input of a UNet. We implement two versions of this model: one where the layer feeding into the loss function is connected to both the output of the Ш-Net and the UNet, which we call ШЦ-Net-Deep, and one where the loss is connected only to the last layer of the whole network.

D. Data Augmentation

In our experiments, we use several schemes to increase the number of images to train our models, since 100 images are not enough to generalize to the test set with high probability. Below are the schemes we implement for our final submission.

1) Rotated Patch Extraction: This augmentation is only used for the Basic CNN model. We split the dataset into 90-10 training-validation images and sample patches with rotations of $0, \frac{\pi}{12}, \frac{\pi}{6}, \frac{\pi}{4}, \frac{\pi}{3}, \frac{5\pi}{12}, \frac{\pi}{2}$ of size 64x64x3 from the training set. We use Poisson disc sampling to prevent highly correlated samples, and generate these patches during the training.

2) Rotation: For the rest of our models, we rotate a given image by $0, \frac{\pi}{6}, \frac{\pi}{3}, \frac{\pi}{2}$ and pad the empty parts with reflections to help the network be more rotation invariant.

3) Translation: We randomly translate an image by $x, y \sim \mathcal{U}(-16, 16)$, bilinearly interpolate and use reflection padding to reduce the effect on translations on network inference.

4) Brightness Scaling: We scale the brightness of an image by a factor of 0.8 in all channels to make the network globally brightness invariant.

5) Zooming and Tiling: We scale a given image by a factors of $0.5, 0.75, 1.00, 1.50$ and put this small image in the middle of an empty image with the same size as the original image. We then tile the empty regions using this small image. This helps the network to also learn finer details, and be more robust in cases where there are variations in the sizes of roads.

E. Loss Functions

To evaluate our classifiers, we use three loss functions and compare their performances in terms of F1 scores and classification accuracy.

1) Mean Squared Error: We use this loss function with our basic CNN model, which is given for a single image by

$$MSE = \frac{1}{N}\sum_i (y_i - f(x_i))^2 \quad (1)$$

where x_i and y_i denote the $i'th$ pixel and f denotes the prediction function.

2) Binary Cross Entropy Loss: This loss function is used with our baseline logistic regression, and in all deconvolutional networks, which is given for a single image by

$$H = -\frac{1}{N}\sum_i \left(y_i \log(f(x_i)) + (1 - y_i)\log(1 - f(x_i))\right) \quad (2)$$

3) Dice Loss: This is used for UNet and its variants and is given for a single image by

$$DL = 1 - \frac{2\sum_i y_i f(x_i)}{\sum_i (y_i^2 + f^2(x_i))} \quad (3)$$

F. Training Details and Datasets

We perform experiments on two datasets generated by the original 100 image dataset. In one dataset, which we call the small dataset, we generate 10 images per image by augmentation (0.75, 1.00 zoom & tiling, and the five rotations given in the augmentation section), and use the rest of the dataset as validation. The other dataset, which we call the large dataset, we generate 15 images per image (0.75, 1.00 zoom & tiling, and the five rotations) and split this set into training and validation. We train each classifier on each of these datasets (except for basic CNN, for which we generate a patch-based dataset) for 150 epochs in total using an ADAM[6] optimizer. For the first 100 epochs, we use a learning rate of $3e-4$ and for the rest of the epochs we use a learning rate of $3e-5$ and use weight regularization/decay of $1e-6$ in the first 100 epochs and $1e-7$ in the last 50 epochs. For baseline, basic CNN, and UNet, we perform the training on an NVIDIA 1660S, which roughly takes three hours to complete for each model, for the small dataset. For the large dataset and the Ш-Net and ШЦ-Net models, we perform the training on an NVIDIA RTX 3090. For the models trained on NVIDIA 1660S, we use a batch size of 1 due to memory constraints, and for NVIDIA RTX 3090, we use a batch size of 16 for Ш-Net and 8 for ШЦ-Net, again due to memory constraints. We save our models in every epoch and use an early stopping scheme based on the validation loss to reduce overfitting. We do not use a k-fold cross validation scheme, simply because training takes too long. Instead, we randomly choose the training set and hope the data generalizes well.

G. Experimental Setup

We perform the experiments involving the simplest models first. We use our results from the simple experiments to construct the experiments in the next step.

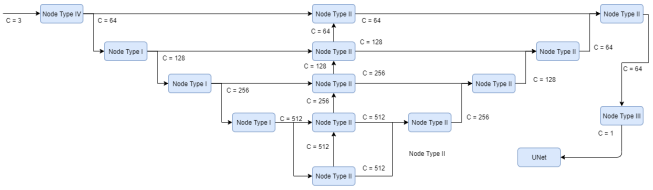


Figure 3: A complete pipeline of our IIIIJ-Net. Using the input and output channels for each node. UNet part of the model is shown as a single node.

Name	Input Channels	Output Channels	Description
Type I	N	N/2	Two Consecutive Convolution Layers
Type II	N	N, N/2, N/4	Two Consecutive Convolutions followed by an Up-Convolution
Type III	64	1	A single Convolutional layer, creates the input for the UNet
Type IV	3	64	A single Convolutional layer

Figure 4: Different types of nodes in our model with their brief description.

1) Baseline Model: The baseline model involves training a logistic regressor over patches of size 16x16. We do not deviate from statistical tradition, and maximize the likelihood of observing pixel labels under a sigmoid probability model, which leads to binary cross entropy. We train the model for 150 iterations, and note the accuracy, precision, recall, and F1 measures. We use the first 90 images without any augmentation as our training set, and use the rest for validation. This experiment is meant to be the control group.

2) Basic CNN: For the basic CNN model, we employ the scheme of considering a patch around a pixel of interest. These experiments are meant to establish CNNs as a more suitable tool than logistic regression for this classification problem. We use pixel-wise MSE loss to evaluate the performance of this model. As its dataset, we extract 90000 patches from the training set using Poisson disk sampling of the 90-10 split, and use the rest of the images as validation.

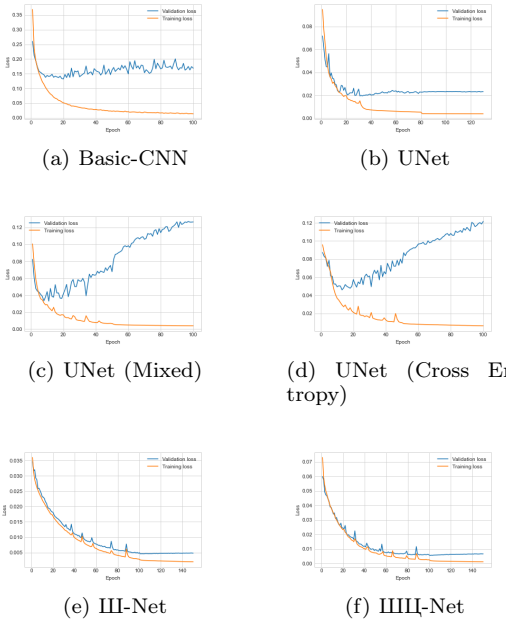
3) UNet: We assume that the inputs to our barebones UNet are whole images of size 320x320, and we train the network parameters according to images of these sizes. We resize an input image, regardless of its size, to 320x320, and rescale the output back into original size of the image for inference. We evaluate the performance of three loss functions in this experiment, and determine which one works best with fully convolutional networks for this dataset. Apart from using binary cross-entropy loss and dice loss, we also consider a third joint loss, parameterized by $\lambda \in (0, 1)$, given as

$$L_{joint}(x, y) = (1 - \lambda)H(x, y) + \lambda DL(x, y) \qquad (4)$$

with $\lambda = 0.2$ and $\lambda = 0.8$ to see if the joint loss results in a better classifier. Depending on the experiments on the small dataset, we pick the best loss function and train the UNet on the large dataset.

4) III-Net: To demonstrate the performance of III-Net, we use the loss with the best performance obtained from UNet experiments. We again assume that inputs are of size 320x320, and follow a similar scheme in evaluating network predictions as in UNet. We train this network only on the larger dataset as described previously and compare the performance of the new architecture with 'vanilla' UNet. To prevent overfitting, we add extra dropout layers with $p = 0.5$

5) IIIIJ-Net: We perform two experiments with IIIIJ-Net. The first experiment is exactly the same as III-Net's experiment, although with IIIIJ-Net. For the second experiment, we enable deep supervision and connect the III-layer of the network to the loss function (i.e. we use IIIIJ-Net-Deep). We perform this experiment to compare both variants of IIIIJ-Net to III-Net, and by extension to the rest of the models.



(a) Basic-CNN (b) UNet

(c) UNet (Mixed) (d) UNet (Cross Entropy)

(e) III-Net (f) IIIIJ-Net

Figure 5: Plots of validation and training loss vs. epochs. Notice that III-Net and IIIIJ-Net tend not to overfit on the training dataset in 150 epochs while other models tend to overfit.

III. Results and Discussion

The experiment with the baseline model takes an insignificant amount of time and yields very poor clas-

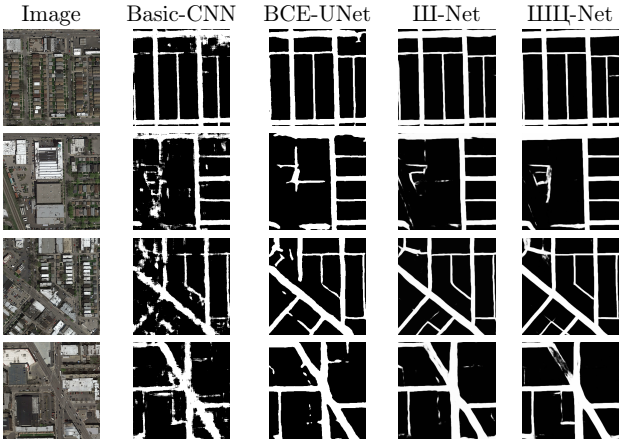


Figure 6: A comparison of sample outputs generated by different models for various images in the dataset

sification accuracy and F1 Score on the validation set, comparable to that of a coin toss.

Basic-CNN model performed better even with our naive choice of MSE loss, surpassing baseline logistic regression significantly. With 16 batch size, the training took 2 hours on the NVIDIA 1660S and used 4GB of VRAM. After 100 epochs with $1e-4$ learning rate with ADAM, the baseline model achieved an F1 score of 0.912 on the validation set and 0.857 on the test set. This model has a particularly slow inference time, as it takes 608x608 evaluations of the neural net per image. It takes around an hour to evaluate the test set.

All UNet variants performed better on the test set in comparison to Basic-CNN on the test set in terms of F1 score. Minimizing Dice-Loss performed the worst, minimizing the binary cross entropy loss performed the best, and the joint loss performed in between (Figure 7 contains the average metrics for the Joint-Loss experiments). We conclude that the dataset we generated is not skewed towards one of the classes, and attribute the inferiority of Dice-Loss to the balancedness of classes. Using UNets, we obtained an F1 score of 0.878 on the test set after 150 epochs. The training took around 3 hours on the NVIDIA 1660S, and with a batch size of 1, 5GB VRAM was required. Inference of the test set took 21.678 seconds, as one evaluation of the network is required for one image.

For the Ш-Net experiment, we used cross-entropy loss, as cross-entropy loss performed better than other losses in the previous experiment. After 150 epochs, Ш-Net surpassed UNet on the test set, achieving an F1 score of 0.907. It took 3 hours to train on the NVIDIA RTX 3090 with batch size 16, and 24GB VRAM was required. Inference of the test set took 7.21 seconds. The added regularization (dropout layers) on top of UNet improved classification accuracy.

For the deep variant of ШШЦ-Net, we performed the same experiment and obtained slightly worse results on the test set (same F1 score, 0.1% inferior accuracy compared to Ш-Net). On the validation set, however, it performed much better than any other model, since gradients being propagated through the Ш-Net component might have allowed the network to get a better fit on the data in the same number of epochs. It took 4 hours to train on the NVIDIA RTX 3090 with batch size 8, and 24GB VRAM was required. Inference of the test set took 8.39 seconds.

Among all models we've performed controlled experiments with, the normal variant of ШШЦ-Net performed the best in the test set, slightly above ШШЦ-Net-Deep and Ш-Net, with an F1 score of 0.908. It took 4 hours to train on the NVIDIA RTX 3090 with batch size 8, and 24GB VRAM was required. Inference of the test set took 8.41 seconds.

Category	Precision	Recall	F1 (Validation)	Accuracy (Validation)	F1 (Test)	Accuracy (Test)
Logistic Regression	0.317	0.681	0.433	0.502	-	-
Basic CNN	0.916	0.908	0.912	0.968	0.857	0.924
UNet (Dice Loss)	0.847	0.894	0.870	0.930	0.863	0.922
UNet (Joint Loss)	0.858	0.967	0.872	0.932	0.866	0.924
UNet (Cross Entropy)	0.840	0.891	0.864	0.927	0.878	0.934
Ш-Net	0.971	0.966	0.968	0.984	0.907	0.950
ШШЦ-Net (Deep)	0.984	0.980	0.982	0.994	0.907	0.949
ШШЦ-Net	0.977	0.968	0.972	0.986	0.908	0.950

Figure 7: A comparison of sample outputs generated by different models for various images in the dataset.

IV. Conclusion

UNet achieves good results in image segmentation tasks [5]. We experimented with UNet and provided two extensions to the architectures, which we call Ш-Net and later ШШЦ-Net. Our experiments showed that our ШШЦ-Net achieves slightly better results at the cost of significant training time. Our model shares most of UNet's strengths, such as its ability to be used for images of arbitrary size. Further improvements to this network could be to fit a model using additional augmentations over the initial dataset, or replace all nodes with residual convolutions; however, we were limited by time and hardware constraints. We provide the full implementation details and our trained models in our repository[1] in the form of a run.py file for our best submission, and in notebooks for the reproduction of our experiments, visualization, training, and model evaluation.

[1] https://github.com/uensalo/epfl-cs433-p2

References

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in Advances in Neural Information Processing Systems 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[2] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," Nature, vol. 585, p. 357–362, 2020.

[3] S. Van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu, "scikit-image: image processing in python," PeerJ, vol. 2, p. e453, 2014.

[4] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," Nature Methods, vol. 17, pp. 261–272, 2020.

[5] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," 2015.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.