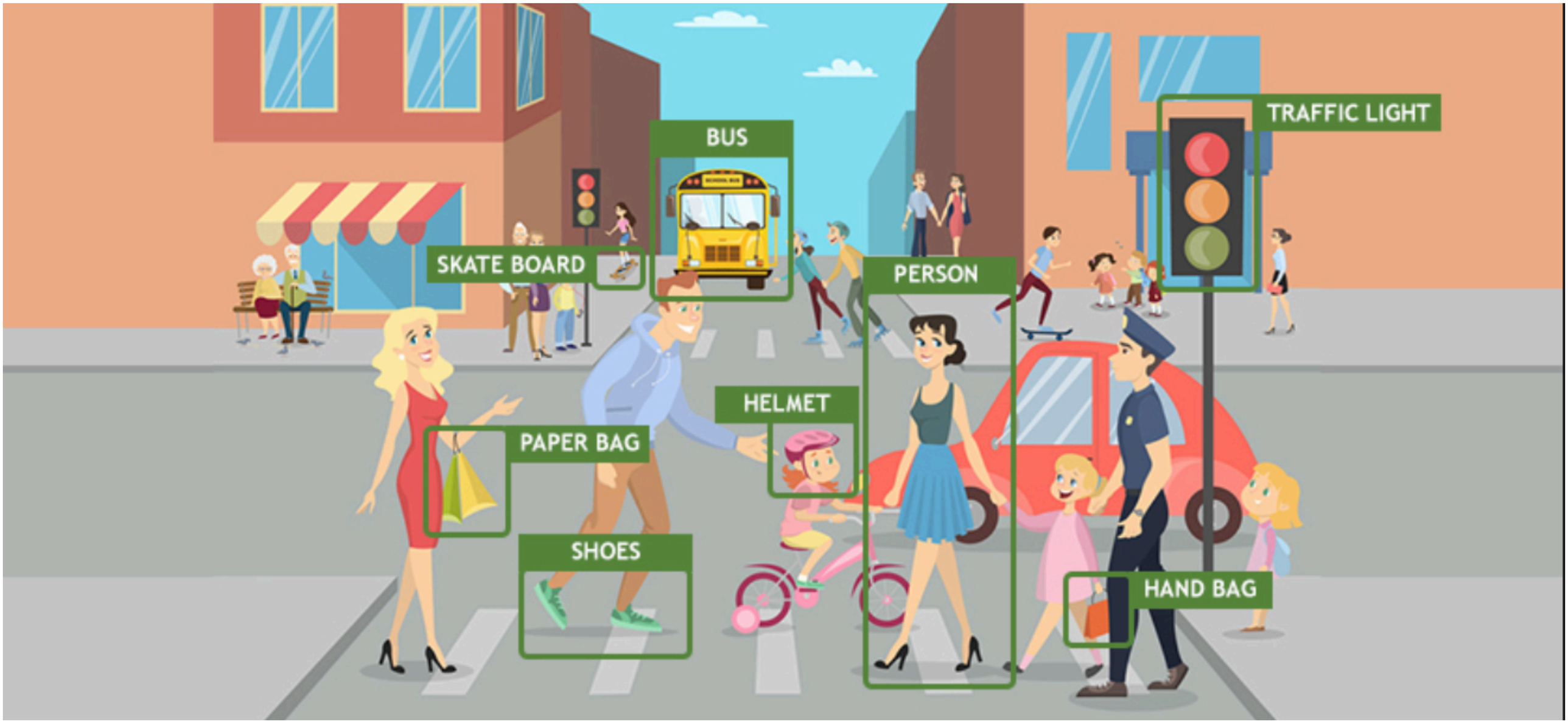
**Mikuláš Formánek - r0736308**

**Irene Volpe - r0740784**

# Computer Vision project



# 3 tasks

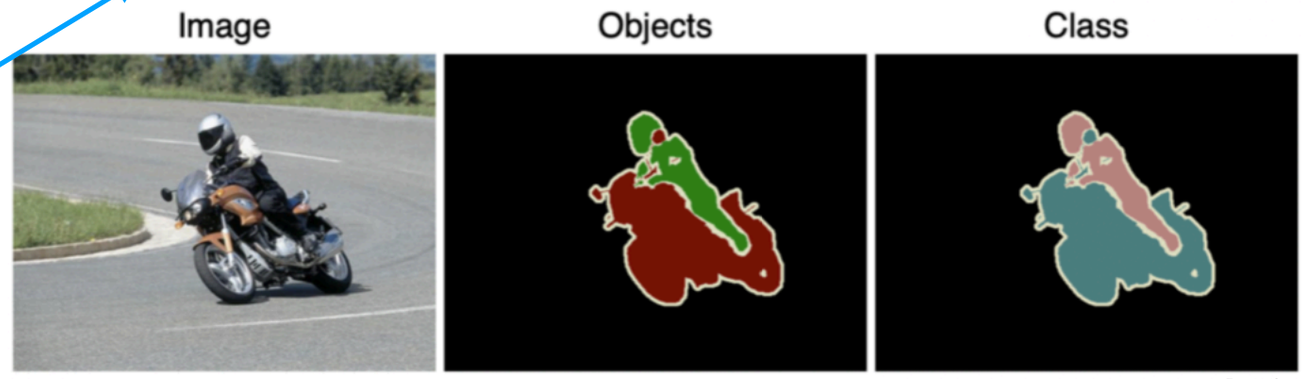
**1) from linear PCA to non-linear convolutional autoencoder**

**2) non-linear convolutional autoencoder: image classification**

**3) segmentation: from image classification to pixel wise classification**


# The dataset

PASCAL VOC-2009 dataset contains colour images of various scenes with different object classes (e.g. animal: bird, cat, ...; vehicle: aeroplane, bicycle, ...), totalling 20 classes (plot1).



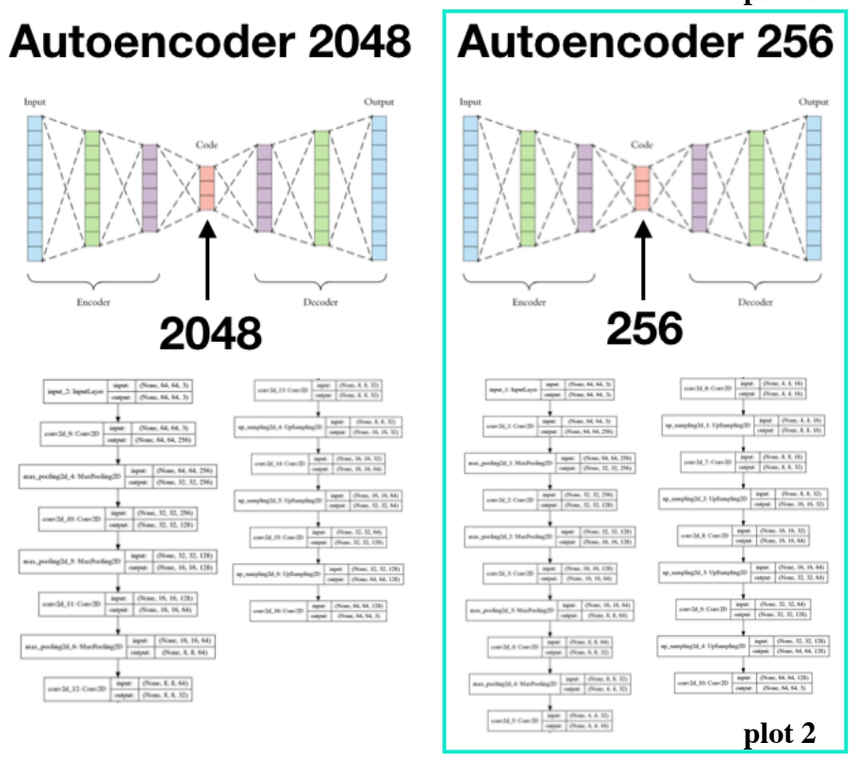

**plot 1**

We did the following operations:.

1) download the training/validation dataset (900 MB TAR-file)

2) build a training and validation set that we will use for this project : we reduced the number of images by selecting only two classes, namely **dogs** and **airplanes** for the first two tasks, then for the segmentation we re-introduced all the 18 remaining classes.

3) for task 1 and 2 we compared two different architectures (plot 2), and used the best one for task 3.

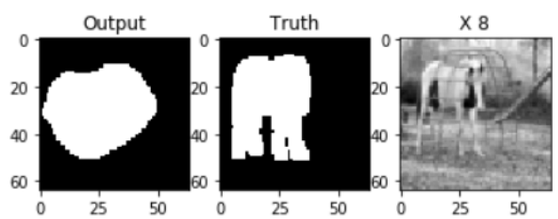



**plot 2**

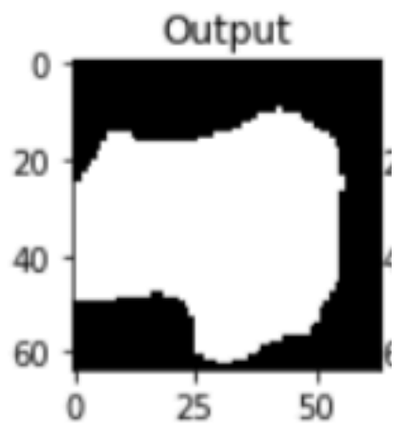
4) For task 3 we first trained 4 different models, with different combinations of: **epochs** (10 and 30), **optimizers** (rmsprop and Adam) **loss functions** (binary cross-entropy and dice coefficient). The best model was trained for 10 epochs using the dice coefficient loss function optimized with the Adam method, adding a batch normalization and dropout layer after each convolution layer (both for encoding and decoding images).

5) Finally, we tried to predict images using the our 4 models trained; we choose the Dice coeffcient and the Intersection over Union as evaluation metrics. As we can see from plot 3, the model with the best training and validation scores also obtained the best dice coefficient and IoU scores, so it was able to draw a better shaped mask for classification tasks.

**plot 3**




Dice Coefficient = 0.64
Intersection over Union = 0.49



**1) from linear PCA to non-linear convolutional autoencoder**

As a first task, we considered two auto-encoders that mimics linear PCA, a method for dimensionality reduction (plot 4). Such auto-encoders contain an input layer, one hidden layer and an output layer.

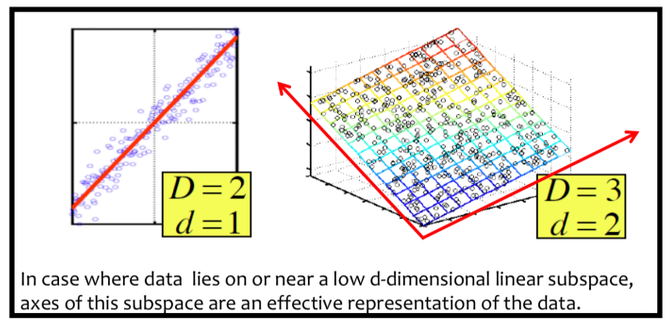
**Differences in results when PCA is framed into a neural network.**

You can use an autoencoder to learn a latent variable representation of the data similar as PCA.

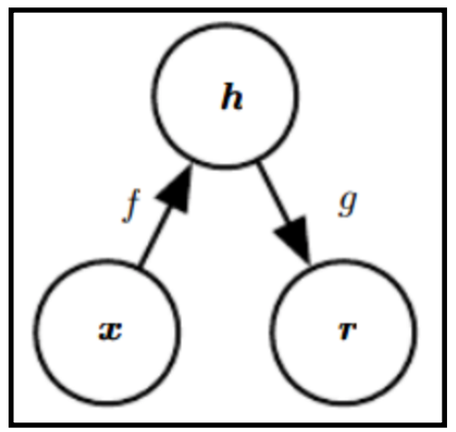
PCA is a linear transformation of the data that finds the direction along which the variance is maximum. As its name suggests, it finds principal components. These components are the structure in the data along which the data is most spread out. So mathematically, PCA is doing following operation: h=Wx.

Where x is the data with higher dimension and h is the projection of the data using W in a lower dimensional space. This transformation maps the data to a space with uncorrelated features (take first k components showing high variance). The same functionality can also be achieved using a neural network. For example, autoencoders are essentially used for this purpose only. The neural networks actually find an affine transformation of the data, if no non-linearity is used for the activation (which is our case, since classes are non linear separable); if we add a linear or an identity activation function, we will get only a linear transformation out of it. After the linear transformation, neural nets minimize the reconstruction error and hence this kind of network will be equivalent to PCA.

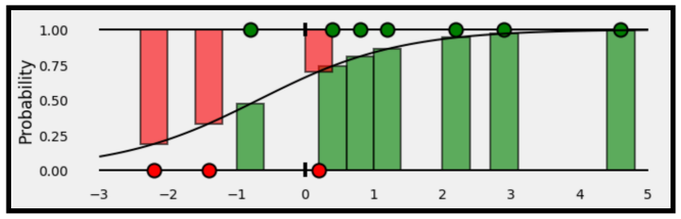
Plot 5 is an autoencoder. It learns to reconstruct the input data (x). Such network learns an identity mapping from x to r and it works as follows: h=f(x) is the latent variable representation of the input r = g(x) = g(f(x)). It is the reconstruction of the input from the latent representation. From the above equation, h is a latent variable representation of x in a low dimensional space. Both the encoder and the decoder are the linear operations and as long as we are using a linear activation function, the output will be an affine transformation of the input $h = W^T x+b$. That means both the encoder and the PCA are the linear operations.




**plot 4**




**plot 5**




**plot 6**

But how can we claim that both PCA and autoencoders are the same kind of projections. If we can show that autoencoder is minimizing the variance between the latent and the exact distribution of data then we are done. Let's see it mathematically: let the loss function of the autoencoder be L(x,g(f(x))), where the loss function is penalizing the network when the distribution g(f(x)) is dissimilar to the distribution of x. In deterministic case, it can be a squared loss otherwise it can be KL-divergence. It simply means that the network is an unbiased estimator which is minimizing the variance between the two distributions. Hence PCA and autoencoder are doing the same job.

In order to diminish the between classes entropy we need to maximize the log-likelihood. plot 6: bars represent the **predicted probabilities** associated with the corresponding **true class** of each point. For this classification task binary crossentropy seems a stronger loss function for maximizing the log-likelihood.

# Autoencoder 2048

## 1) from linear PCA to non-linear convolutional autoencoder

**3) Deep training using Keras:**

size (training set) =
517 images 64,64,3

size (validation set) =
539 images 64,64,3

batch_size = 32,

epochs = 10,

loss function: MSE

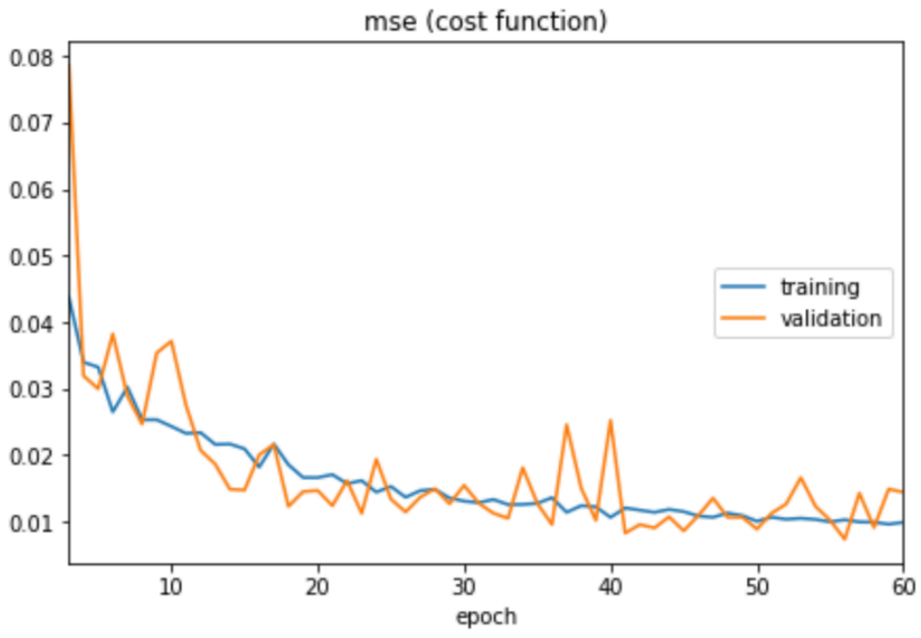
optimizer: rmsprop

MSE (training) = 0.025

MSE (validation) = 0.042

MSE (reconstruction_error) = 0.04

MAE (reconstruction_error) = 0.17



# Autoencoder 2048

## 2) non-linear convolutional autoencoder: image classification

size (training set) = 917 images 64,64,3

size (validation set) = 139 images 64,64,3

epochs = 50,

loss function: binary-crossentropy (log)

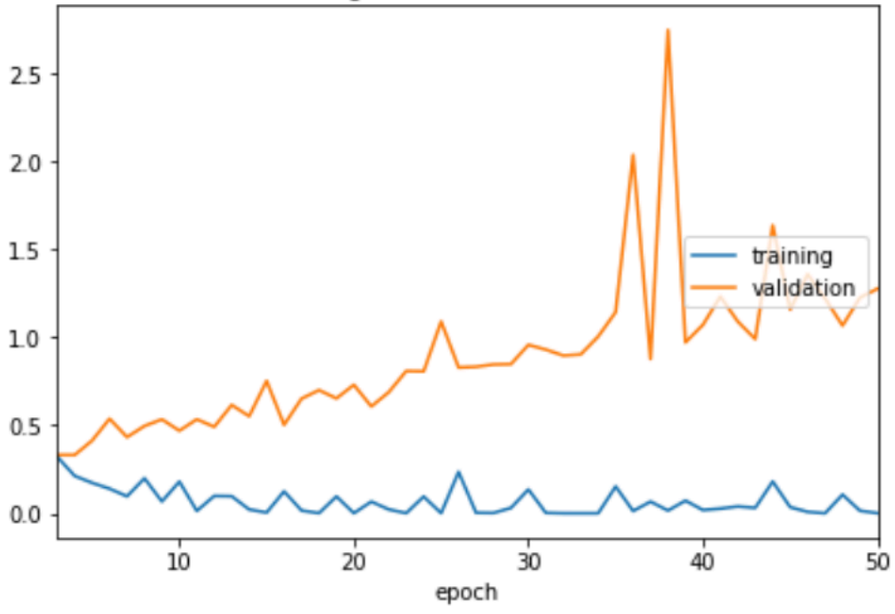
optimizer: rmsprop

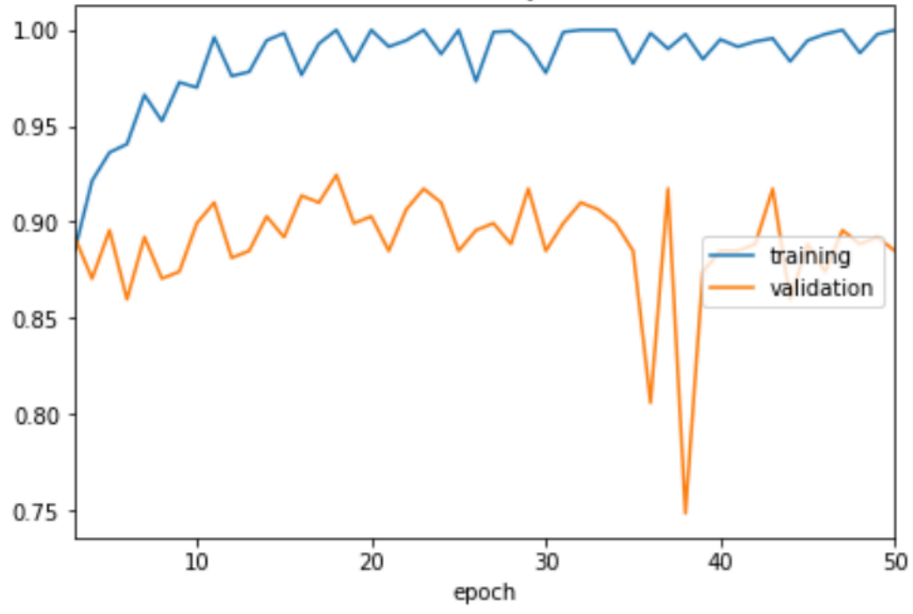
logloss (training set) = 0.001

logloss (validation set) = 1.276

Accuracy (training set) = 1.00

Accuracy (validation set) = 0.885



# Autoencoder 256

1) from linear PCA to non-linear convolutional autoencoder

**Deep training using Keras:**

size (training set) = 517 images 64,64,3

size (validation set) = 539 images 64,64,3

batch_size = 32,

epochs = 10,

loss function: MSE

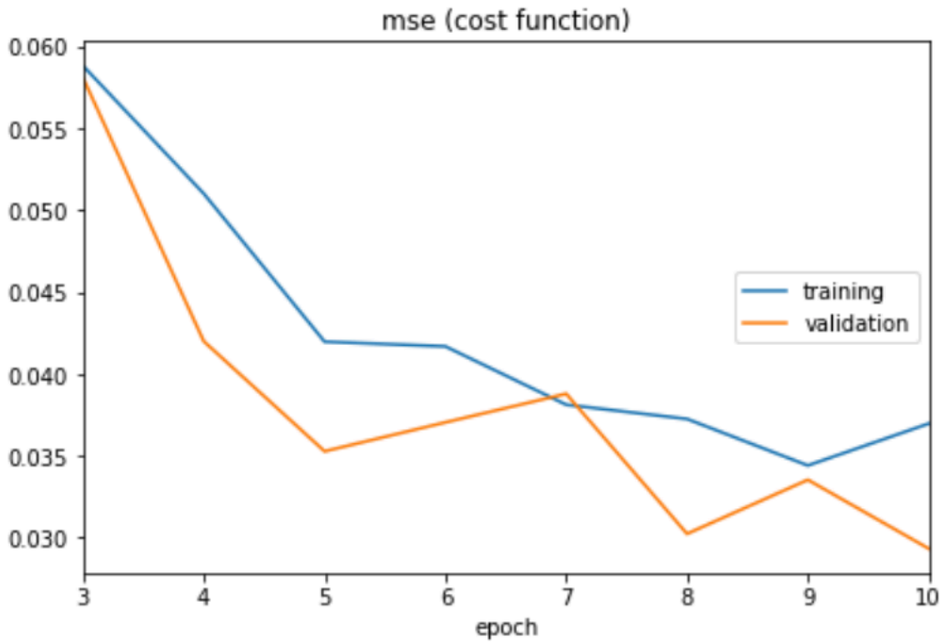
optimizer: rmsprop

MSE (training) = 0.037

MSE (validation) = 0.029

MSE (reconstruction_error) = 0.03

MAE (reconstruction_error) = 0.13




visualize the learned coding space and compare this with what you had before.

**Compared to Autoencoder 2048:**

MSE (training) = 0.025

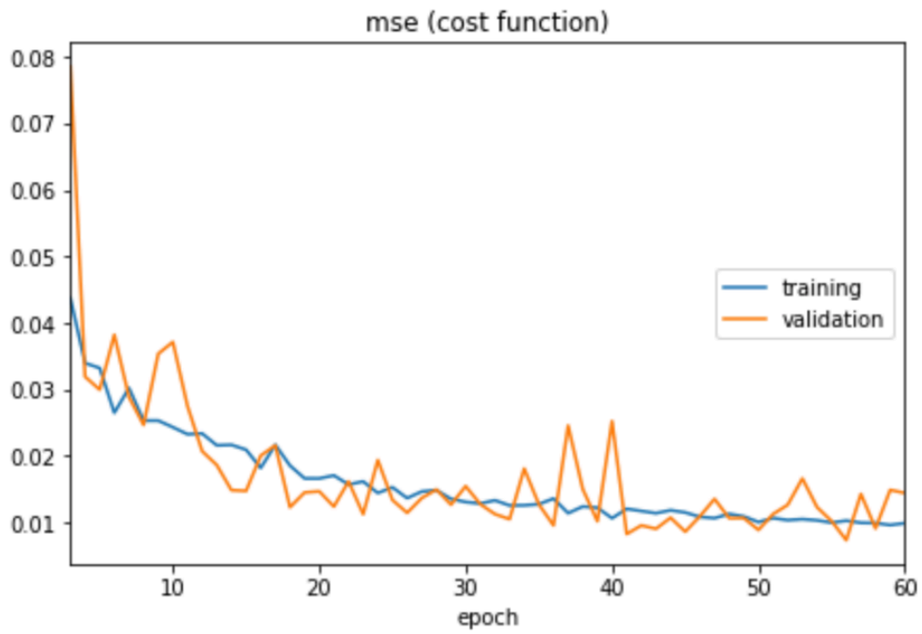
MSE (validation) = 0.042

MSE (reconstruction_error) = 0.04

MAE (reconstruction_error) = 0.17

better on the training yet weaker on the validation

slightly weaker in reconstructing images



# Autoencoder 256

## 2) non-linear convolutional autoencoder: image classification

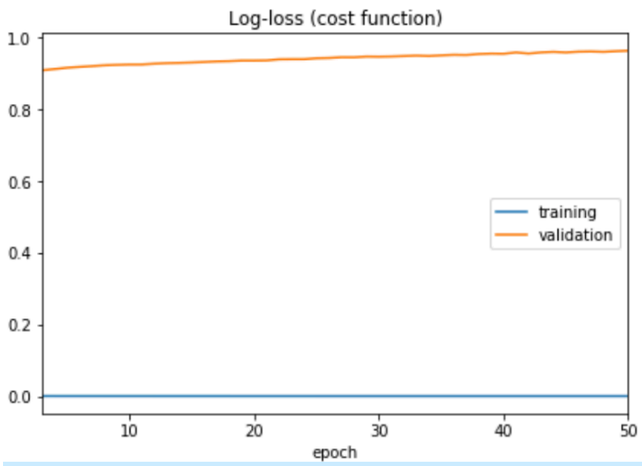
size (training set) = 917 images 64,64,3

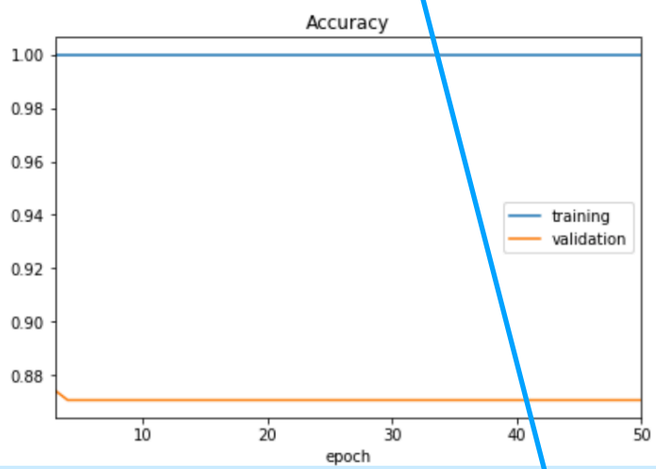
size (validation set) = 139 images 64,64,3

**1** epochs = 50,
loss function: binary-crossentropy (log)
optimizer: **sgd**

logloss (training set) = 0.00
logloss (validation set) = 0.96

Accuracy (training set) = 1.00
Accuracy (validation set) = 0.871

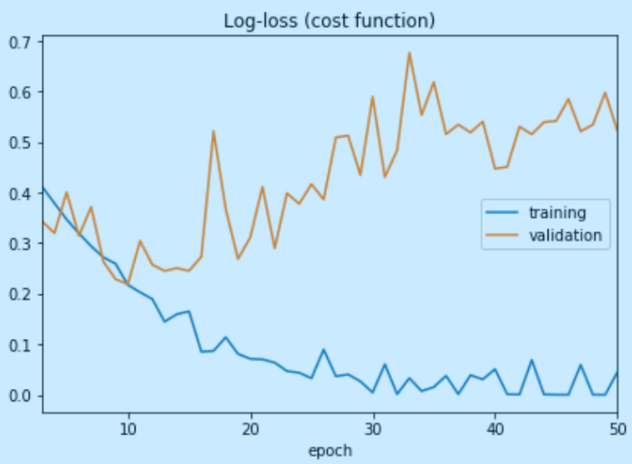


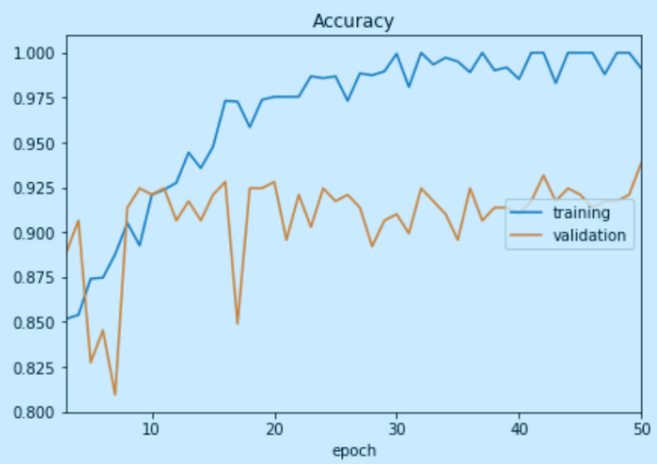




**2** epochs = 50,
loss function: binary-crossentropy (log)
optimizer: **rmsprop**

logloss (training set) = 0.045
logloss (validation set) = 0.522

Accuracy (training set) = 0.991
Accuracy (validation set) = 0.939




visualize the learned coding space and compare this with what you had before.

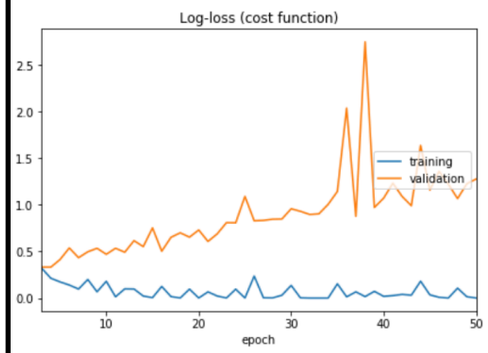
**Compared to Autoencoder 2048:**

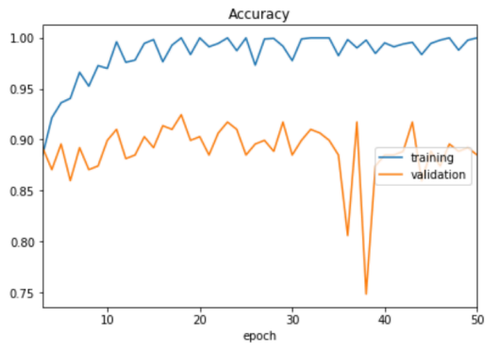
logloss (training set) = 0.00
logloss (validation set) = 1.276

better on the training yet **weaker on the validation**

Accuracy (training set) = 1.00
Accuracy (validation set) = 0.885

better on the training yet **weaker on the validation**




**Conclusions:**

We trained two classifiers using binary crossentropy as loss function, then logloss and accuracy as evaluation metrics.

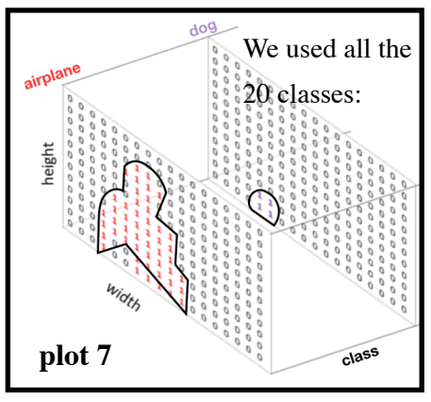
Comparing the best classifier trained with 256 variables (rmsprop as optimizer) with the classifier trained with 2048 variables we can see that according to both logloss and Accuracy metrics the former performs better on the validation sets than the latter. We therefore proceeded to the segmentation part employing the 256 variable trained non-linear convolutional autoencoder.

# Autoencoder 256

## 3) segmentation: from image classification to pixel wise classification

In image segmentation we label specific regions of an image according to what's being shown.
The goal is to predict class labels for each pixel in the image.
We're not separating *instances* of the same class; we only care about the category of each pixel: if you have two objects of the same category in your input image, the segmentation map does not inherently distinguish these as separate objects (*instance segmentation* models do).

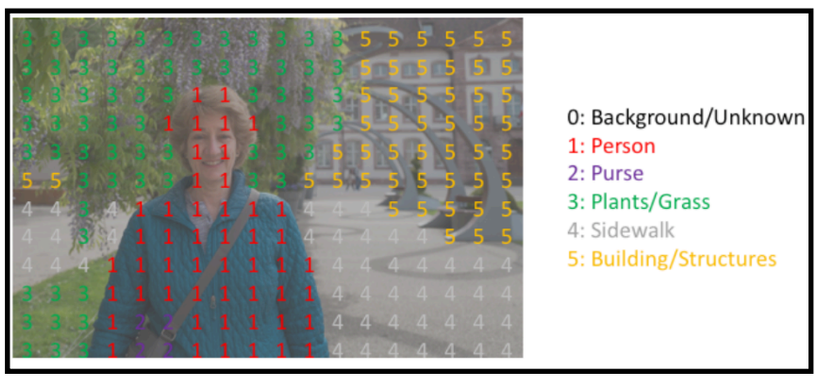
### Goal

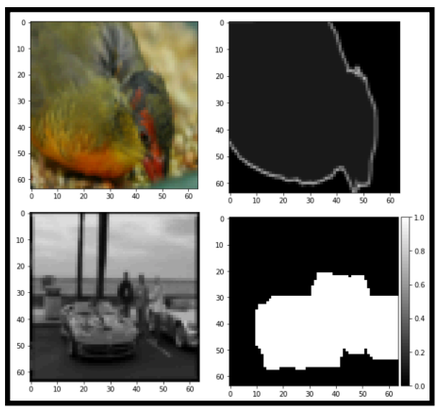

Our goal is to take our RGB image (h,w,3) and output a segmentation map where each pixel contains a class label represented as an integer (h,w,1) (**plot7).**

Note: we have low-resolution prediction map because the input is the encoded vector of instances as output of the autoncoder (**plot 8-11).**
In reality, the segmentation label resolution should match the original input's resolution.

We created our **target** by one-hot encoding the class labels.
A prediction can be collapsed into a segmentation map (as shown in the first image) by taking the argmax of each depth-wise pixel vector.
We can easily inspect a target by overlaying it onto the observation (**plot 12**).

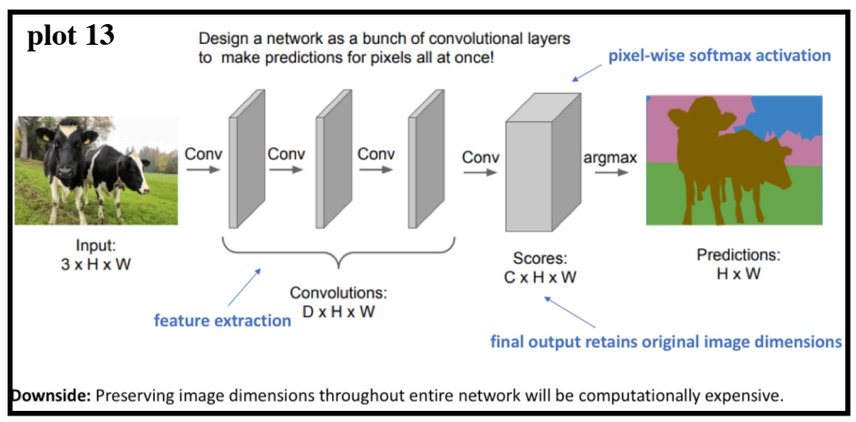



**plot 12:** observation and class labels overlayed.



**plot 8**: **input**= RGB training image (first class)
**plot 9**: labeled mask (first class)
**plot 10**: greyscale training images (second class)
**plot 11**: predicted mask (second class) —-> based on such mask, **output** is most probable class the image is assigned to.

When we overlay a *single channel* of our target (or prediction), we refer to this as a **mask** which illuminates the regions of an image where a specific class is present (**plot 11).** The downside is that preserving image dimensions throughout the network would me computationally expensive, so we make a network deep and work at a lower spatial resolution for many of the layers (**plot13**).



# Autoencoder 256

## 3) segmentation: from image classification to pixel wise classification

One popular approach for image segmentation models is to follow an **encoder/ decoder structure** where we *downsample* the spatial resolution of the input, developing lower-resolution feature mappings which are learned to be highly efficient at discriminating between classes, and the *upsample* the feature representations into a full-resolution segmentation map (**plot 14**).

There are a few different approaches that we can use to *upsample* the resolution of a feature map. Whereas pooling operations downsample the resolution by summarizing a local area with a single value (ie. average or max pooling), "unpooling" operations upsample the resolution by distributing a single value into a higher resolution.

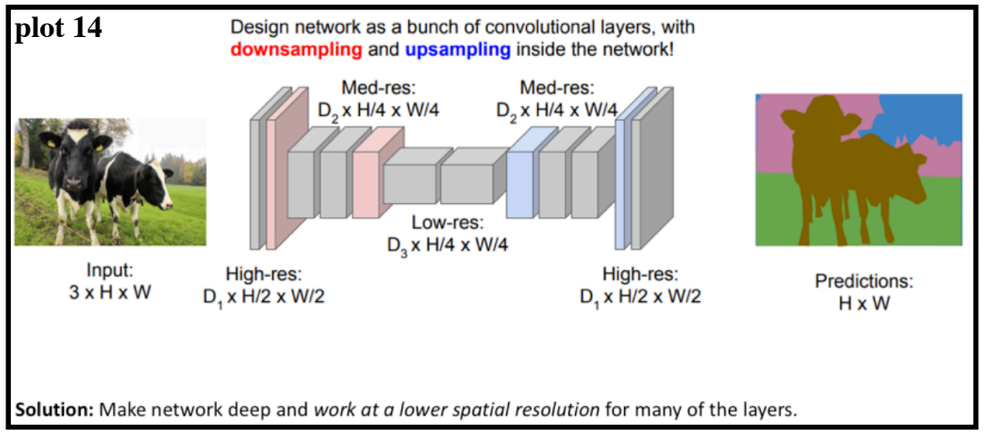
However, **transpose convolutions** (**plot 15**) are by far the most popular approach as they allow for us to develop a *learned upsampling*. Whereas a typical convolution operation will take the dot product of the values currently in the filter's view and produce a single value for the corresponding output position, a transpose convolution essentially does the opposite. For a transpose convolution, we take a single value from the low-resolution feature map and multiply all of the weights in our filter by this value, projecting those weighted values into the output feature map. The overlapping values are simply added together. A simplified 1D example of upsampling through a transpose operation is shown in **plot 16**.

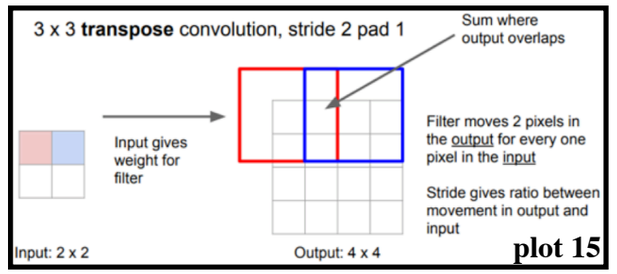
The approach of using a "fully convolutional" network trained end-to-end, pixels-to-pixels for the task of image segmentation was introduced by Long et al. in late 2014. The paper's authors propose adapting existing, well-studied *image classification* networks (eg. AlexNet) to serve as the encoder module of the network, appending a decoder module with transpose convolutional layers to upsample the coarse feature maps into a full-resolution segmentation map. (**plot 17**). The full network, as shown in **plot 18**, is trained according to a pixel-wise cross entropy loss.

However, because the encoder module reduces the resolution of the input by a factor of 32, the decoder module **struggles to produce fine-grained segmentations** (**plot 19**).

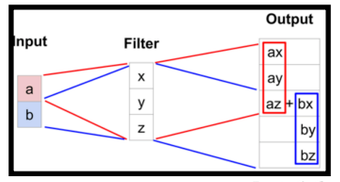
*Semantic segmentation faces an inherent tension between semantics and location: global information resolves **what** while local information resolves **where**... Combining fine layers and coarse layers lets the model make local predictions that respect global structure. — Long et al.*

**plot 14**



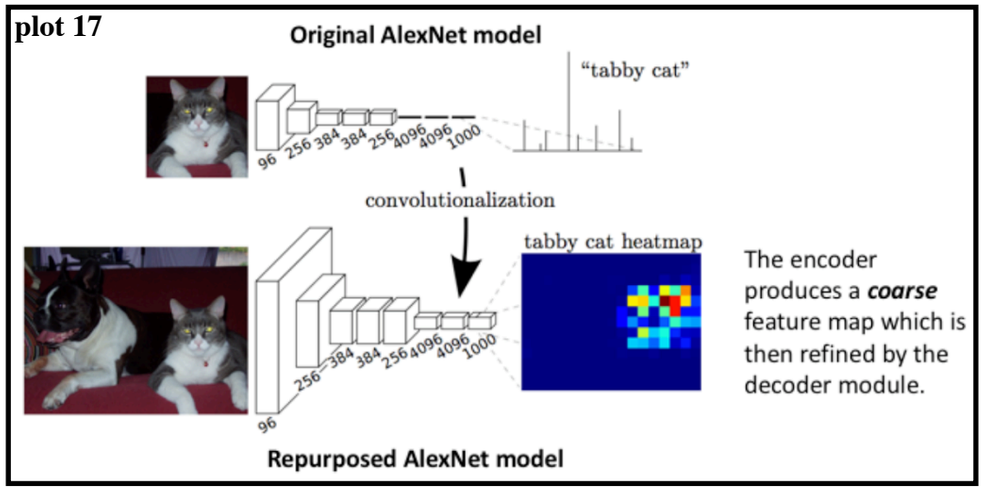



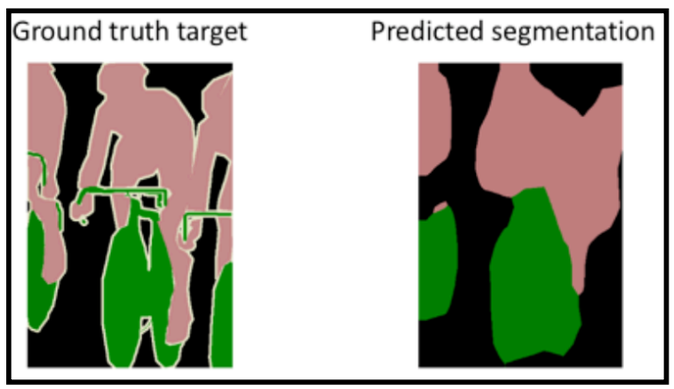

**plot 15**




**plot 16**

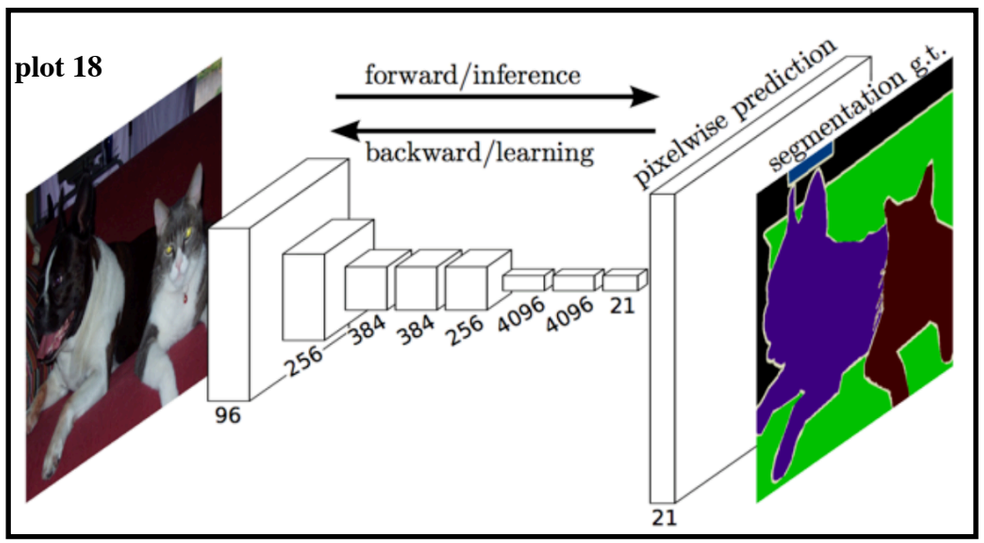
**plot 17**




**plot 19**

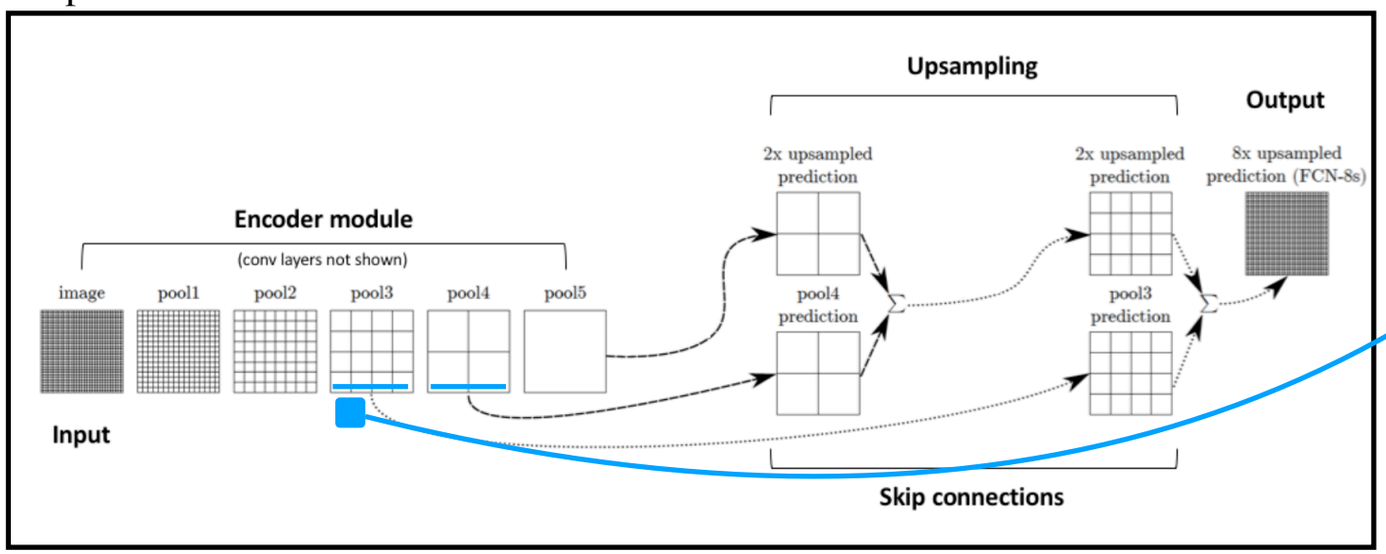
**plot 18**



# Autoencoder 256

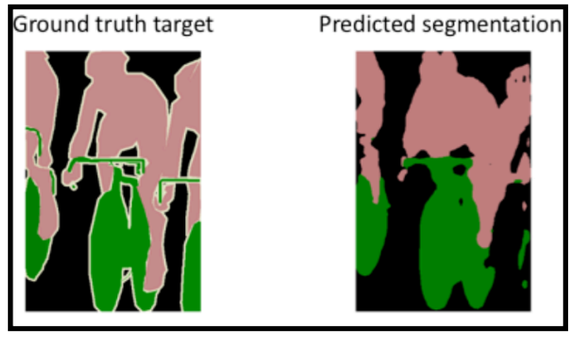
## 3) segmentation: from image classification to pixel wise classification

**plot 20:** this tension can be addressed by upsampling the encoded representations, adding "skip connections" from earlier layers, and summing these two feature maps.




```python
def segment(input_img):
    #encoder
    conv1 = Conv2D(256, (3, 3), activation='relu', padding='same')(input_img)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
    conv2 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool1)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
    conv3 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool2)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
    conv4 = Conv2D(32, (3, 3), activation='relu', padding='same')(pool3)
    pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)
    conv5 = Conv2D(16, (3, 3), activation='relu', padding='same')(pool4)
```

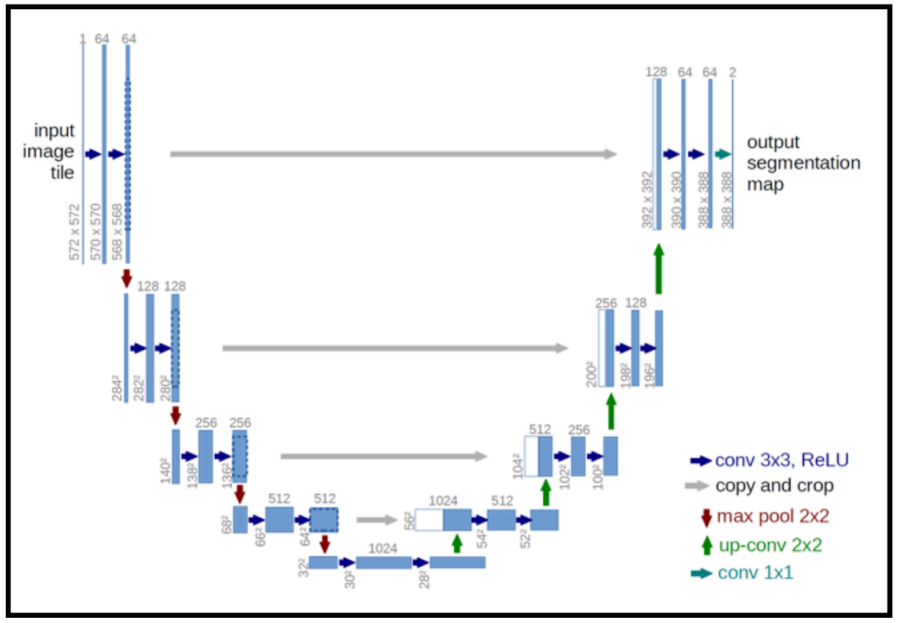
**plot 20**




**plot 21**

**plot 21:** These skip connections from earlier layers in the network (prior to a downsampling operation) should provide the necessary detail in order to reconstruct accurate shapes for segmentation boundaries. Indeed, we can recover more fine-grain detail with the addition of these skip connections.

**plot 22:** Ronneberger et al. improve upon the "fully convolutional" architecture primarily through ***expanding the capacity of the decoder*** module of the network. More concretely, they propose the **U-Net architecture** which "consists of a contracting path to capture context and a ***symmetric*** expanding path that enables precise localization." This simpler architecture has grown to be very popular and has been adapted for a variety of segmentation problems.




**plot 22**

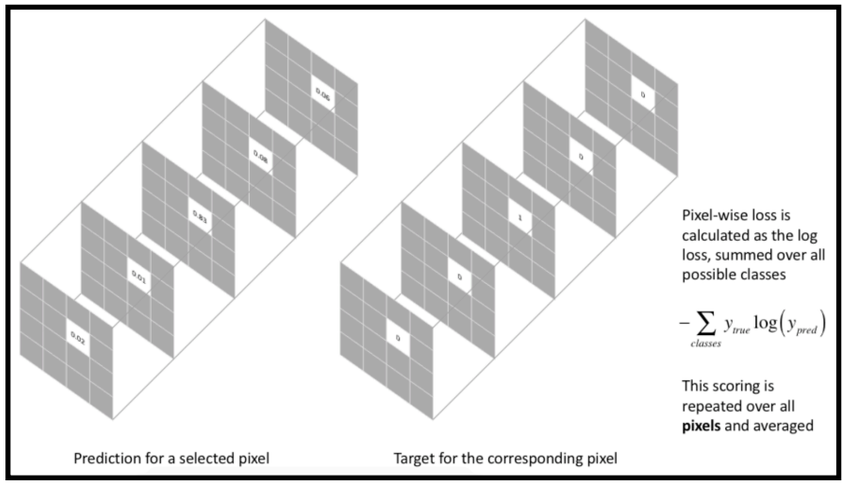
# Autoencoder 256

## 3) segmentation: from image classification to pixel wise classification

We tried 4 different models, with different combinations of:
**epochs** (10 and 30),
**optimizers** (rmsprop and Adam)
**loss functions** (binary cross-entropy and dice coefficient)

## Loss functions description

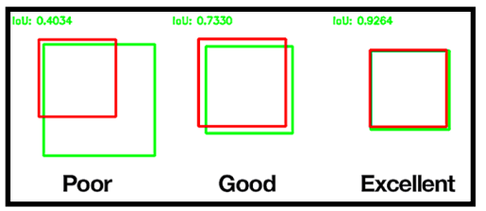
**Binary (pixel-wise) cross entropy loss** is the most commonly used loss function for the task of image segmentation is a. This loss examines *each pixel individually*, comparing the class predictions (depth-wise pixel vector) to our one-hot encoded target vector.
For each of the 20 neurons in the output layer we assign a value, ie the binary cross entropy loss, which is proportional to the negative of the log likelihood of that corresponding class to be the right class; therefore the smaller the value for output neuron 3 the greater the chance the mask presented to the network comes from class 3 of our images dataset.
**Dice coefficient loss** is essentially a measure of overlap between two samples. This measure ranges from 0 to 1 where a Dice coefficient of 1 denotes perfect and complete overlap. The Dice coefficient was originally developed for binary data, and can be calculated as shown in **plot xxxx.** For the case of evaluating a Dice coefficient on predicted segmentation masks, we can approximate $|A \cap B|$ as the element-wise multiplication between the prediction and target mask, and then sum the resulting matrix **(plot xxxy).**
Because our target mask is binary, we effectively zero-out any pixels from our prediction which are not "activated" in the target mask. For the remaining pixels, we are essentially penalizing low-confidence predictions; a higher value for this expression, which is in the numerator, leads to a better Dice coefficient. There's a 2 in the numerator in calculating the Dice coefficient because our denominator "double counts" the common elements between the two sets. In order to formulate a loss function which can be minimized, we'll simply use 1-Dice.
This loss function is known as the soft Dice loss because we directly use the predicted probabilities instead of thresholding and converting them into a binary mask. With respect to the neural network output, the numerator is concerned with the *common activations* between our prediction and target mask, where as the denominator is concerned with the quantity of activations in each mask *separately*. This has the effect of normalizing our loss according to the size of the target mask such that the soft Dice loss does not struggle learning from classes with lesser spatial representation in an image. A soft Dice loss is calculated for each class separately and then averaged to yield a final score.
**The advantage compared to the Binary cross entropy loss function used before is that it retains sensitivity in more heterogeneous data sets and gives less weight to outliers.**




$$Dice(A,B) = \frac{2|AB|}{|A| + |B|}$$

$$Dice = \frac{2 \times TP}{(TP + FP) + (TP + FN)}$$

$$IoU = \frac{TP}{(TP + FP + FN)}$$




**plot 23, plot 24, plot 25,**

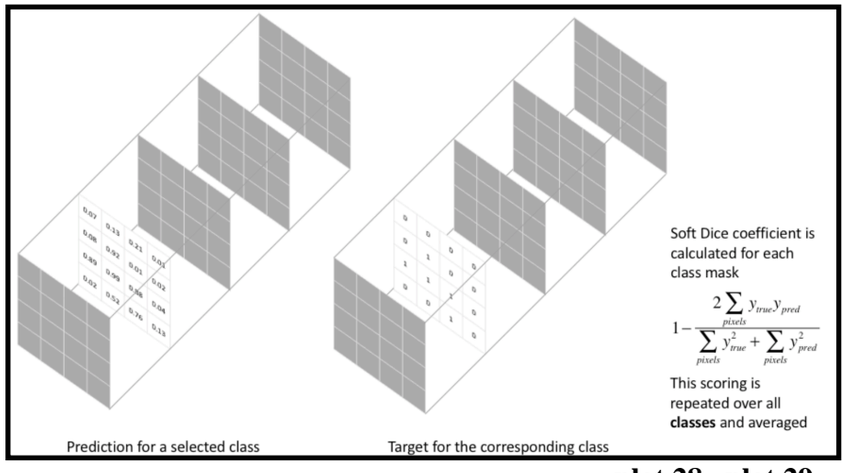
$$|A \cap B| = \begin{bmatrix} 0.01 & 0.03 & 0.02 & 0.02 \\ 0.05 & 0.12 & 0.09 & 0.07 \\ 0.89 & 0.85 & 0.88 & 0.91 \\ 0.99 & 0.97 & 0.95 & 0.97 \end{bmatrix} * \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \xrightarrow{\text{element-wise multiply}} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.89 & 0.85 & 0.88 & 0.91 \\ 0.99 & 0.97 & 0.95 & 0.97 \end{bmatrix} \xrightarrow{\text{sum}} 7.41$$

prediction target

**plot 26, plot 27,**

$$|A| = \begin{bmatrix} 0.01 & 0.03 & 0.02 & 0.02 \\ 0.05 & 0.12 & 0.09 & 0.07 \\ 0.89 & 0.85 & 0.88 & 0.91 \\ 0.99 & 0.97 & 0.95 & 0.97 \end{bmatrix}^{2\ (optional)} \xrightarrow{\text{sum}} 7.82$$

$$|B| = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}^{2\ (optional)} \xrightarrow{\text{sum}} 8$$




**plot 28, plot 29,**

# Autoencoder 256

### 3) segmentation: from image classification to pixel wise classification

## Optimizers description

Optimizers are methods to build upon gradient descent to combat the problem of pathological curvature, and speed up search in the loss function at the same time. These methods are often called "Adaptive Methods" since the learning step is adapted according to the topology of the contour.

**RMSProp** tries to dampen the oscillations by *automatically* adjusting the learning rate (as opposite to the momentum optimizer). More so, it choses a different learning rate for each parameter.

**Plot xx**: exponential average of the gradient squared, yet different for momentum optimizer we're doing it for every parameter. Since we do it separately for each parameter, gradient *Gt* here corresponds to the projection, or component of the gradient along the direction represented by the parameter we are updating.

To do that, we multiply the exponential average computed till the last update with a hyperparameter, represented by the greek symbol *nu*. We then multiply the square of the current gradient with *(1 - nu)*. We then add them together to get the exponential average till the current time step.

So while momentum accelerates our search in direction of minima, RMSProp impedes our search in direction of oscillations.

**Adam** or **Adaptive Moment Optimization** algorithms combines the heuristics of both Momentum and RMSProp.

**Plot xy:** the update equations. Here, we compute the exponential average of the gradient as well as the squares of the gradient for each parameters (Eq 1, and Eq 2). To decide our learning step, we multiply our learning rate by average of the gradient (as was the case with momentum) and divide it by the root mean square of the exponential average of square of gradients (as was the case with momentum) in equation 3. Then, we add the update.

$$For\ each\ Parameter\ w^j$$
$$(j\ subscript\ dropped\ for\ clarity)$$
$$\nu_t = \rho\nu_{t-1} + (1-\rho) * g_t^2$$
$$\Delta\omega_t = -\frac{\eta}{\sqrt{\nu_t + \epsilon}} * g_t$$
$$\omega_{t+1} = \omega_t + \Delta\omega_t$$
$$\eta : Initial\ Learning\ rate$$
$$\nu_t : Exponential\ Average\ of\ squares\ of\ gradients$$
$$g_t : Gradient\ at\ time\ t\ along\ \omega^j$$

**plot xx:** rmsprop

$$For\ each\ Parameter\ w^j$$
$$(j\ subscript\ dropped\ for\ clarity)$$
$$\nu_t = \beta_1 * \nu_{t-1} - (1-\beta_1) * g_t$$
$$s_t = \beta_2 * s_{t-1} - (1-\beta_2) * g_t^2$$
$$\Delta\omega_t = -\eta\frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$
$$\omega_{t+1} = \omega_t + \Delta\omega_t$$
$$\eta : Initial\ Learning\ rate$$
$$g_t : Gradient\ at\ time\ t\ along\ \omega^j$$
$$\nu_t : Exponential\ Average\ of\ gradients\ along\ \omega_j$$
$$s_t : Exponential\ Average\ of\ squares\ of\ gradients\ along\ \omega_j$$
$$\beta_1, \beta_2 : Hyperparameters$$

**plot xxx:** Adam

# Autoencoder 256

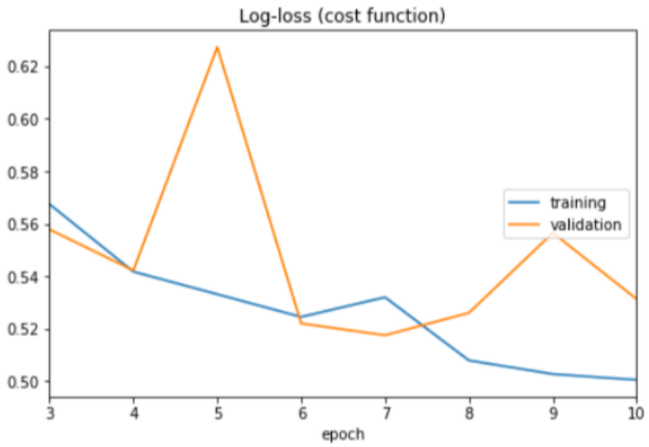
## 3) segmentation: from image classification to pixel wise classification

Dice Coefficient = 0.50
Intersection over Union = 0.34

**1) train**

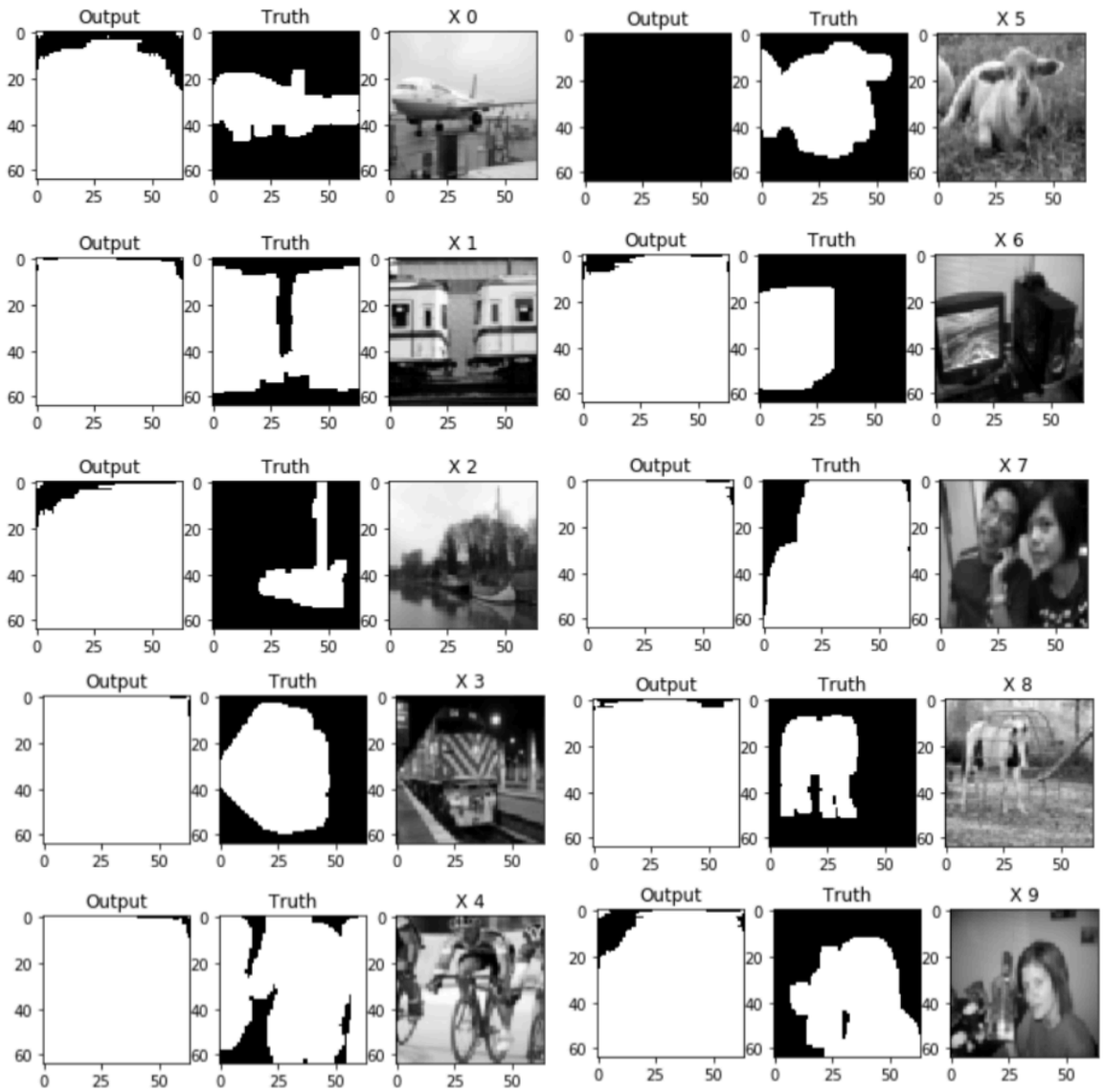
1 input img = (64, 64, 1))
batch_size = 32

**epochs = 10,**
**loss function: binary-crossentropy**
**optimizer: rmsprop**




best valid logloss : 0.518 (epoch 7).

**2) predict**



# Autoencoder 256

## 3) segmentation: from image classification to pixel wise classification

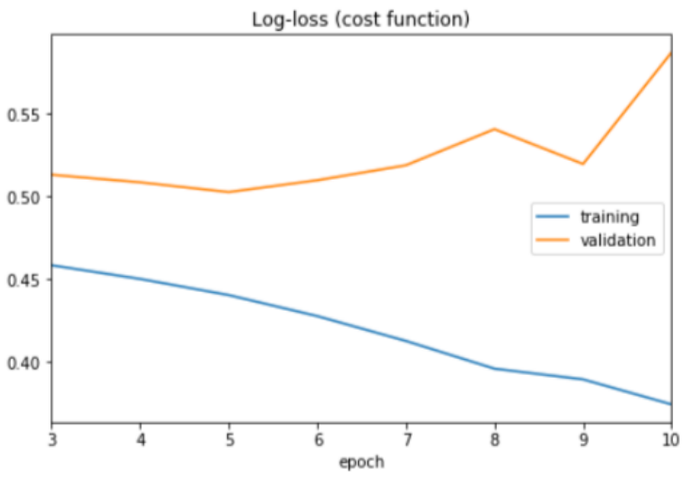
Dice Coefficient = 0.50
Intersection over Union = 0.34

Dice Coefficient = 0.54
Intersection over Union = 0.38

**1) train**

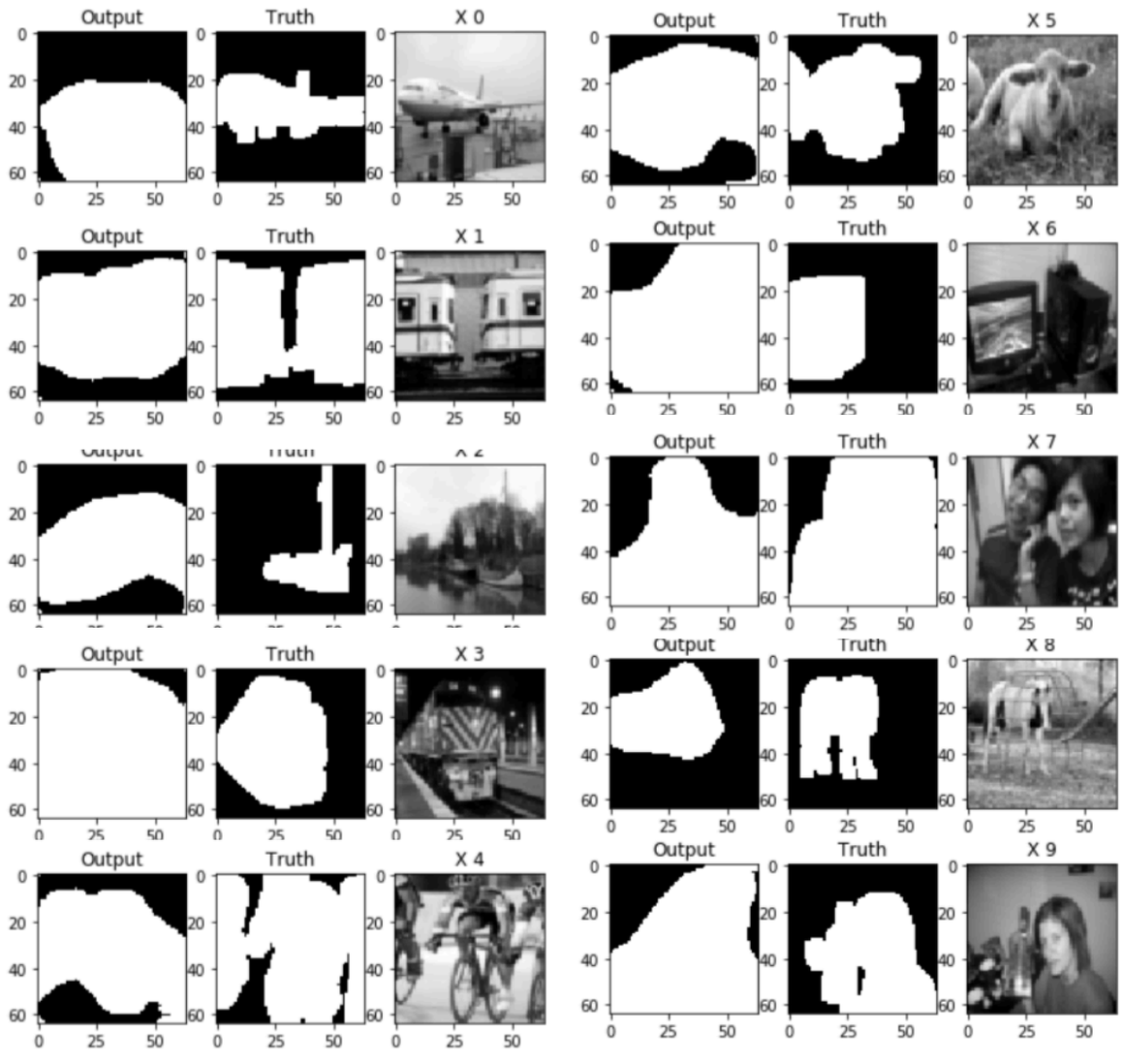
**2** input img = (64, 64, 1))
batch_size = 32

**epochs = 10,**
**loss function: binary-crossentropy**
**optimizer: Adam**




best valid logloss : 0.500 (epoch 5).

**2) predict**



# Autoencoder 256

## 3) segmentation: from image classification to pixel wise classification

Dice Coefficient = 0.50
Intersection over Union = 0.34

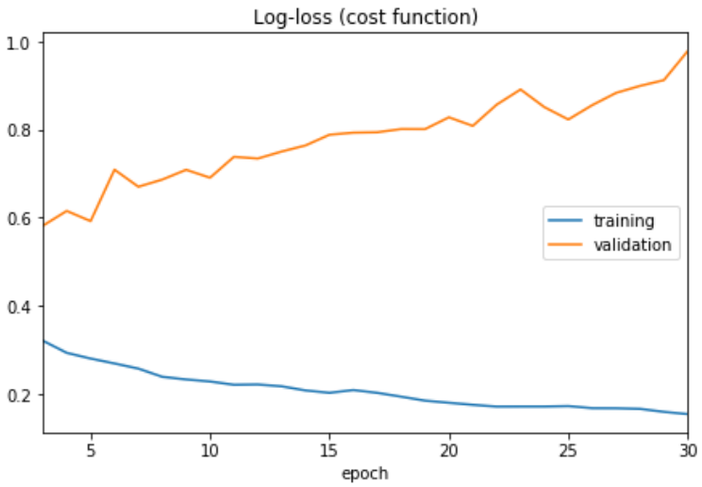
Dice Coefficient = 0.54
Intersection over Union = 0.38

Dice Coefficient = 0.60
Intersection over Union = 0.45

### 1) train

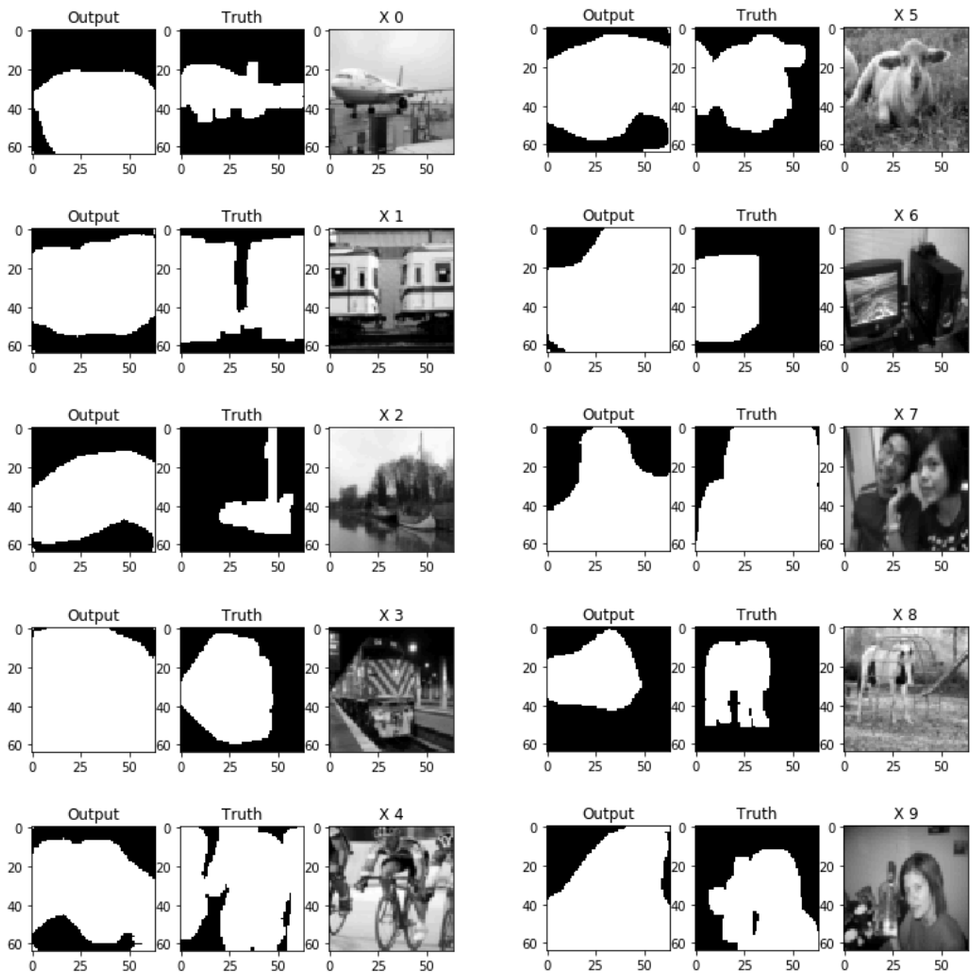
**3** input img = (64, 64, 1))
batch_size = 32

**epochs = 30,**
**loss function: binary-crossentropy**
**optimizer: Adam**




best valid logloss : 0.541 (epoch 4).

### 2) predict



# Autoencoder 256

## 3) segmentation: from image classification to pixel wise classification

Dice Coefficient = 0.50
Intersection over Union = 0.34

Dice Coefficient = 0.54
Intersection over Union = 0.38

Dice Coefficient = 0.60
Intersection over Union = 0.45

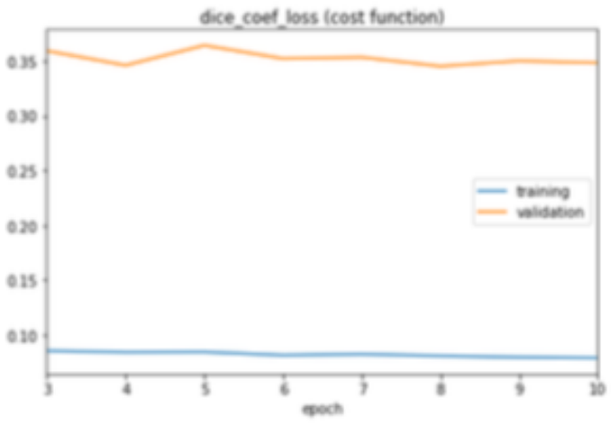
Best both for IoU and Dice metrics

Dice Coefficient = 0.63
Intersection over Union = 0.48

**1) train**

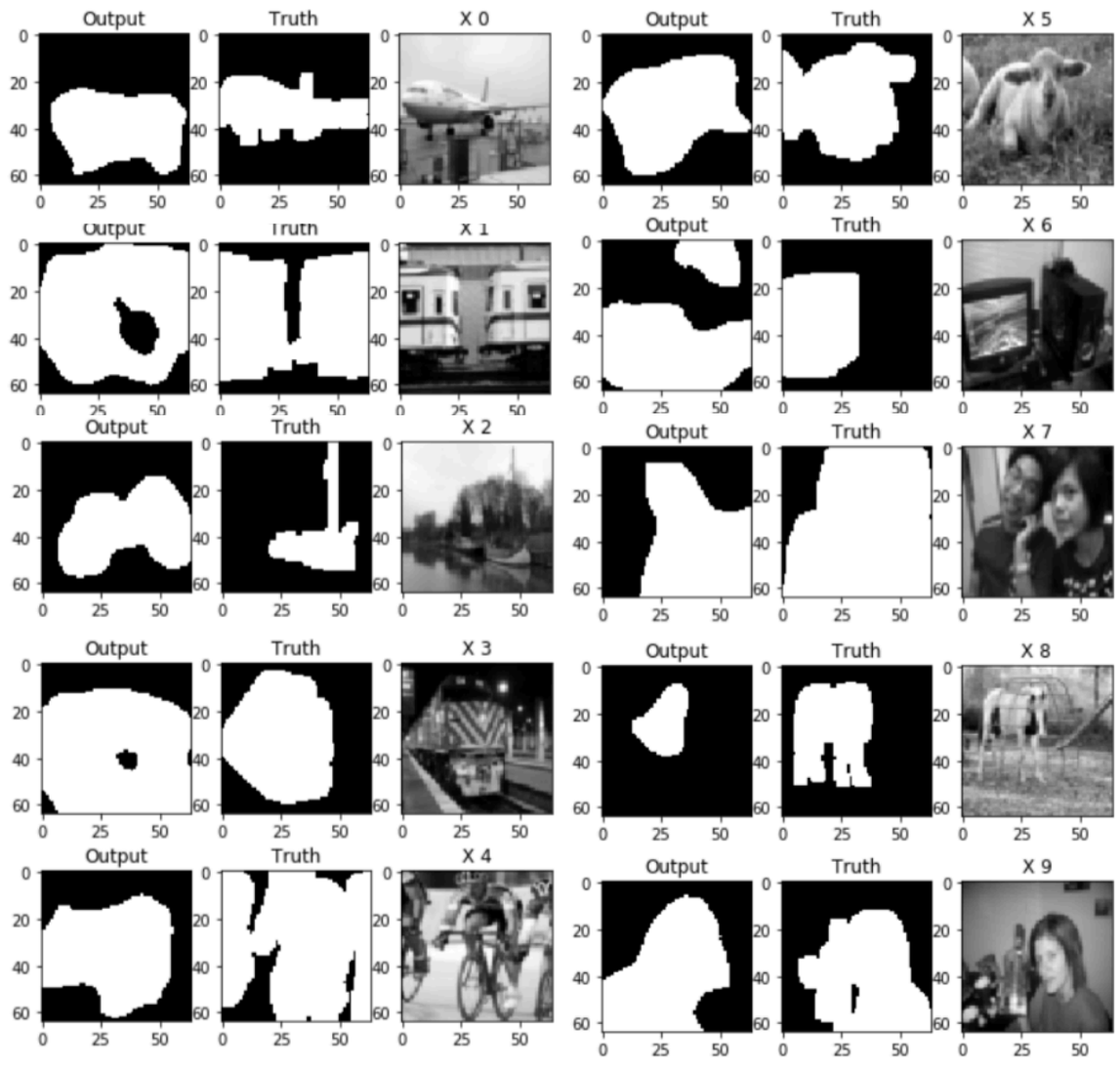
**4** input img = (64, 64, 1))
batch_size = 32

**epochs = 10,**
**loss function: Dice coefficient**
**optimizer: Adam**




best valid logloss : 0.345 (epoch 6).

**2) predict**



# Autoencoder 256

### 3) segmentation: from image classification to pixel wise classification

Adding a batch normalization and dropout layer after each convolution layer (both for encoding and decoding images).

Best both for IoU and Dice metrics

Dice Coefficient = 0.50
Intersection over Union = 0.34

Dice Coefficient = 0.54
Intersection over Union = 0.38

Dice Coefficient = 0.60
Intersection over Union = 0.45

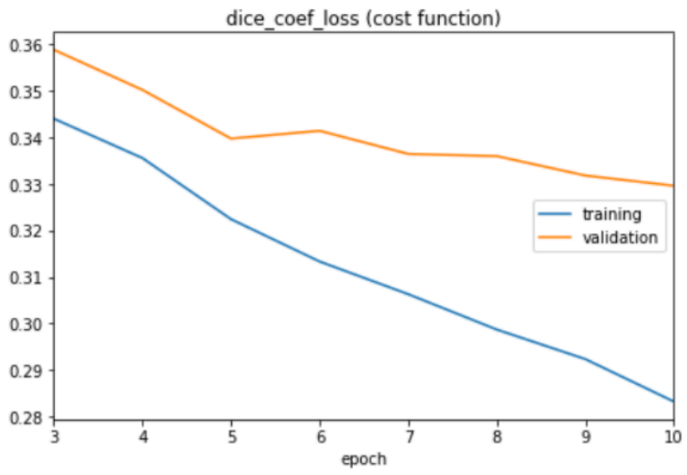
Dice Coefficient = 0.63
Intersection over Union = 0.48

Dice Coefficient = 0.64
Intersection over Union = 0.49

**1) train**

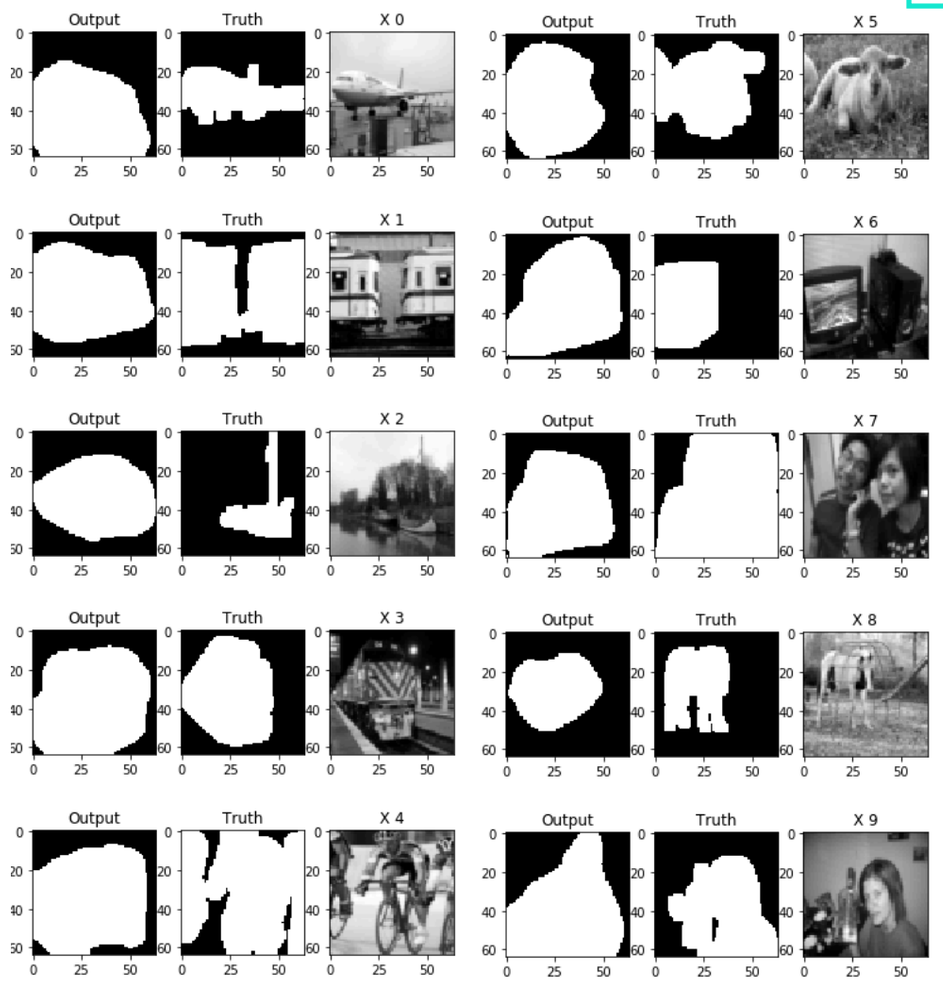
**5** input img = (64, 64, 1))
batch_size = 32

**epochs = 10,**
**loss function: Dice coefficient**
**optimizer: Adam**




best valid logloss : 0.330 (epoch 10).

**2) predict**



# Autoencoder 256

## 3) segmentation: from image classification to pixel wise classification

Adding a batch normalization and dropout layer after each convolution layer (both for encoding and decoding images).

Best both for IoU and Dice metrics

Dice Coefficient = 0.50
Intersection over Union = 0.34

Dice Coefficient = 0.54
Intersection over Union = 0.38

Dice Coefficient = 0.60
Intersection over Union = 0.45

Dice Coefficient = 0.63
Intersection over Union = 0.48

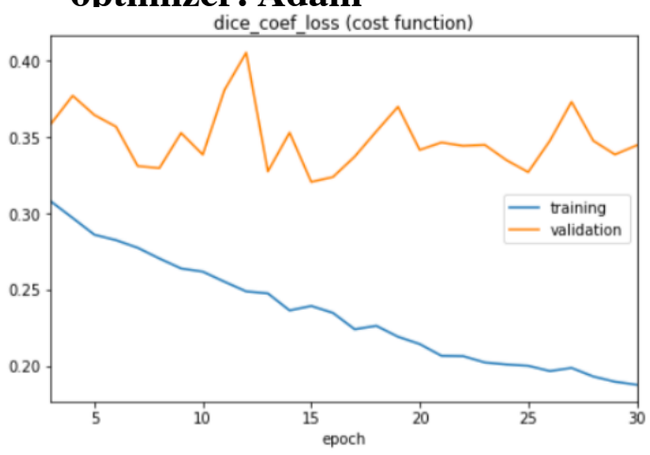
Dice Coefficient = 0.64
Intersection over Union = 0.49

Dice Coefficient = 0.62
Intersection over Union = 0.48

**1) train**

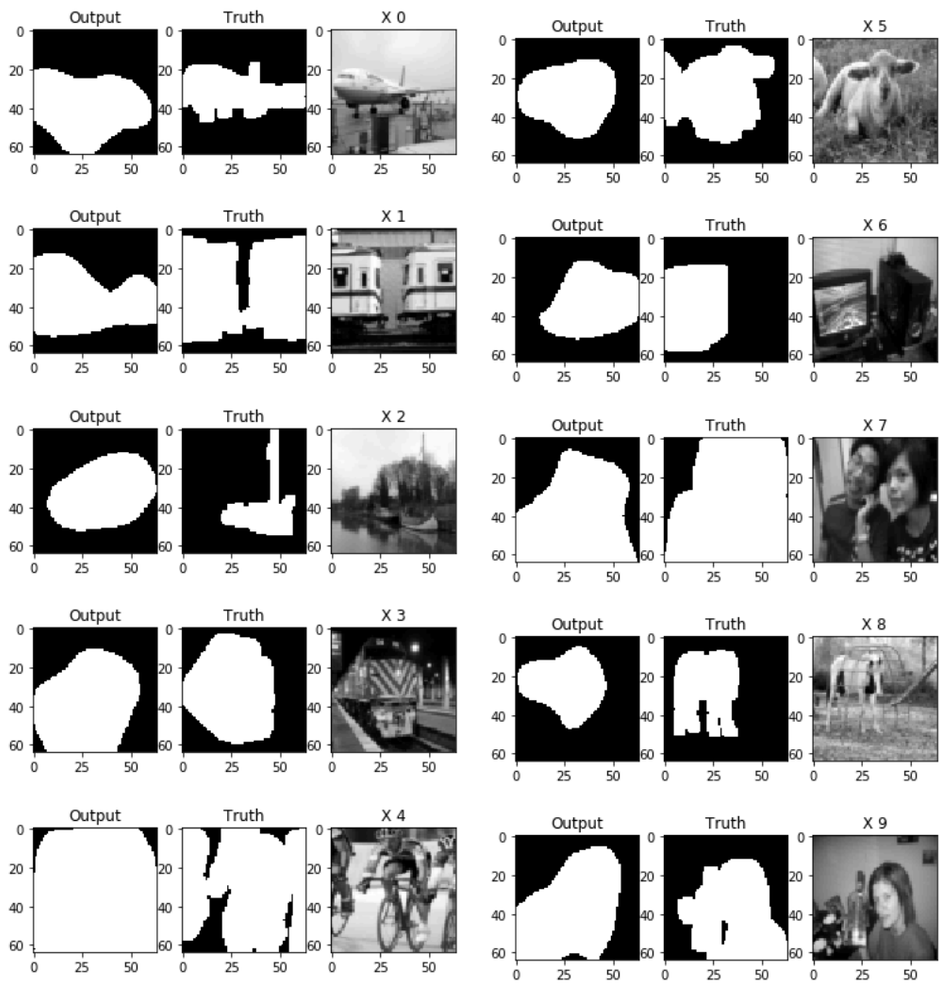
**6** input img = (64, 64, 1))
batch_size = 32

**epochs = 30,**
**loss function: Dice coefficient**
**optimizer: Adam**




best valid logloss : 0.321 (epoch 15).

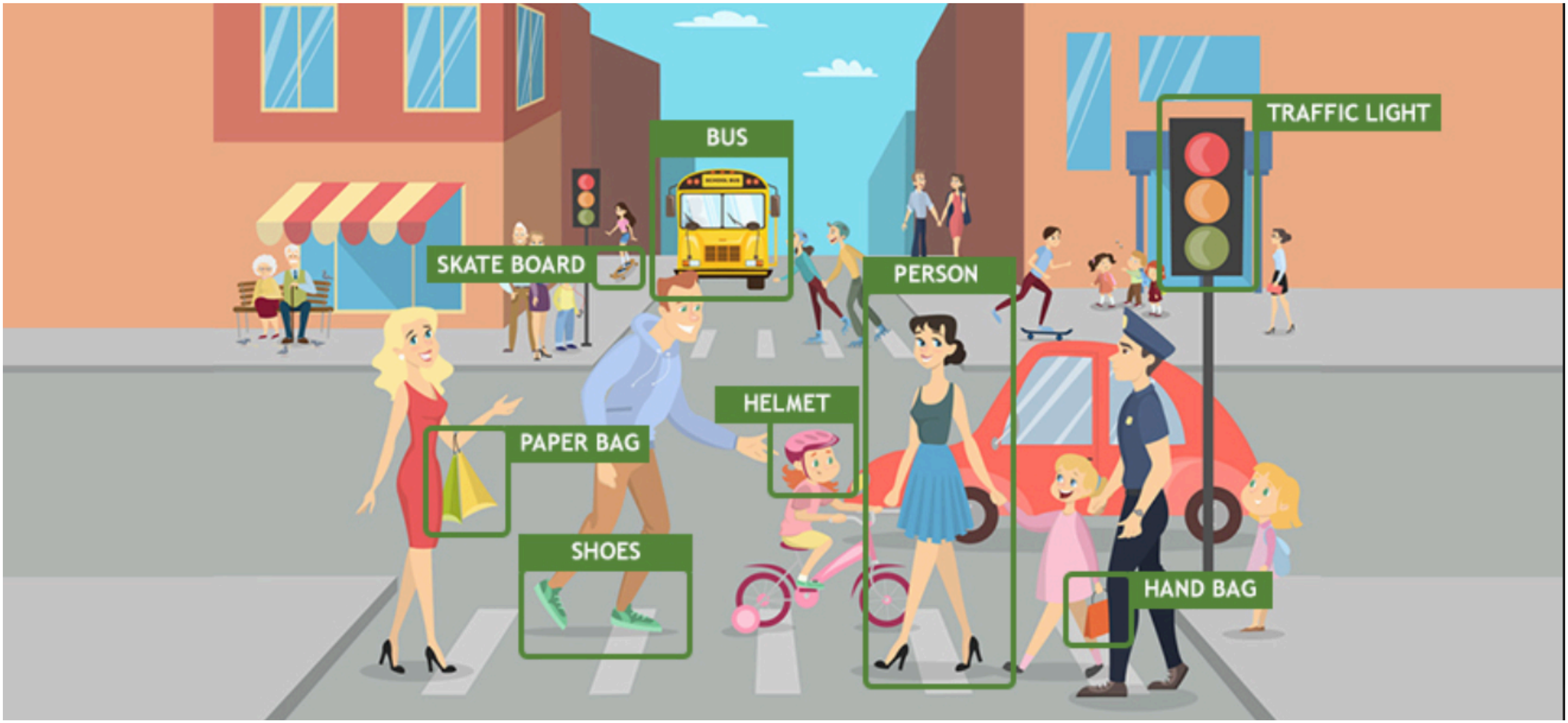
**2) predict**



**Mikuláš Formánek - r0736308**

**Irene Volpe - r0740784**

# Computer Vision project




**Thank you for reading**