

Disparity-Based Collision Avoidance System

May 31, 2016

Deep Learning Project
Skoltech, Spring 2016

- Anastasia Makarova
- Mikhail Usvyatsov
- Mikhail Karasikov
- Daniil Merkulov

1 Introduction

The goal of our project is to predict the distance to the nearest object on the road from a single image. Neural Networks are widely used in Advanced Driver Assistance Systems (ADAS) and this problem has direct application in Camera-Based Forward Collision Alert System.

Ordinary, disparity map is built based on stereo pair and then camera parameters are necessary for depth map building.

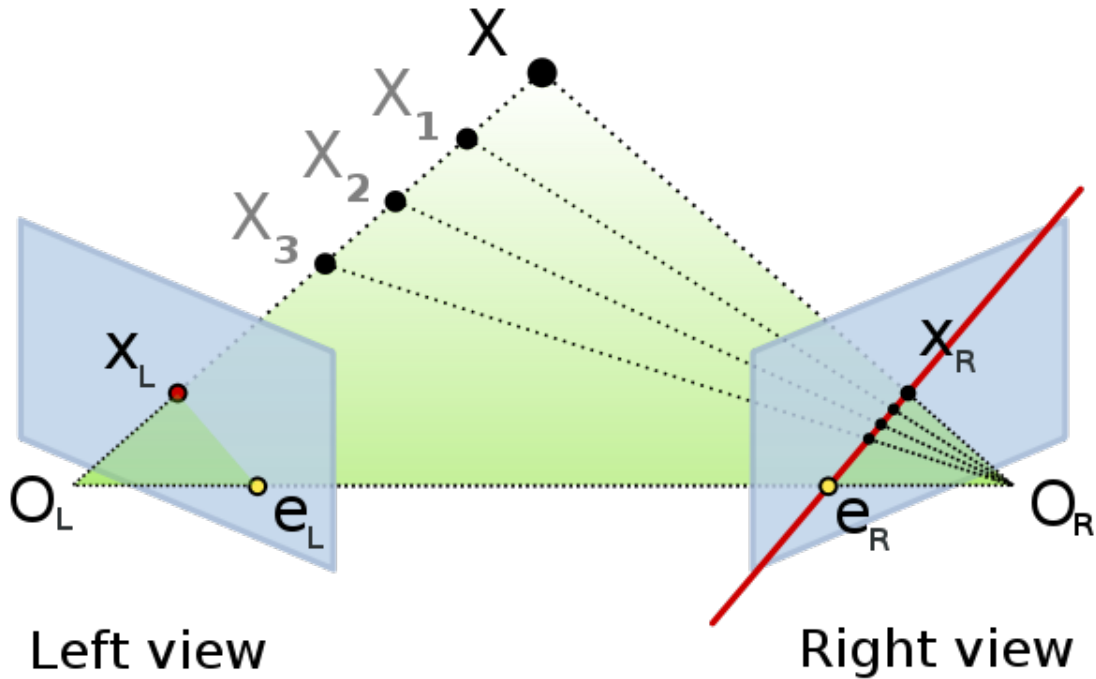


Figure 1:

We use semantic annotated data CityScapes and all intrinsic parameters are known, so the goal was to predict the robust maximum for disparity map, corresponded o the closest object on the road.

As long as we are interested only in objects on the way of the car, we work not only with the entire image, but also with its Region of Interest (RoI).

2 Related work

There are several works related to the whole depth map prediction, based on Deep Learning. In *'Depth Map Prediction from a Single Image using a Multi-Scale Deep Network'*, 2015 authors consider two steps (coarse-scale and fine-scale) NN Structure. The task of the coarse-scale network is to predict the overall depth map structure using a global view of the scene. After taking a global perspective, local refinements are made by fine-scale network.

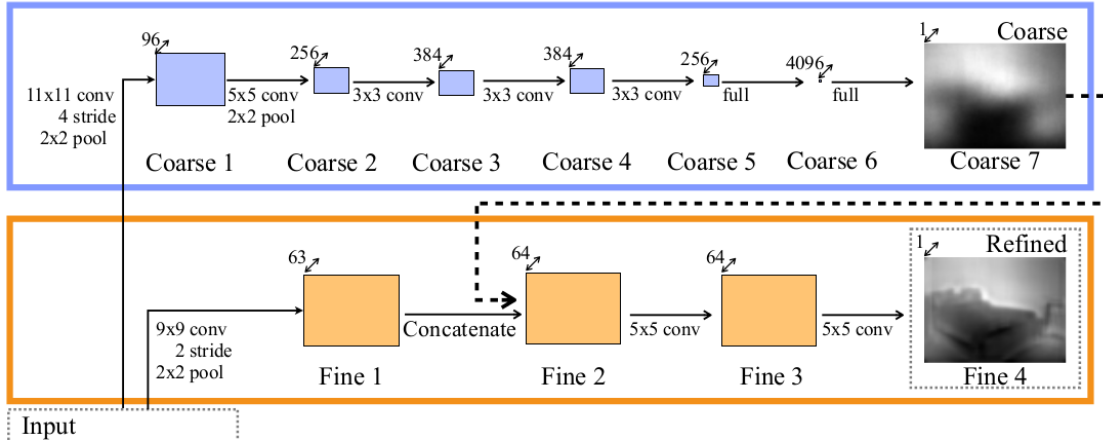


Figure 2:

The highly impressive result motivated us to use deep learning approaches for our task.

3 Problem setting

The mathematical formulation of problem we deal with is the following.

X — RGB pictures

Y — labels

$X^n = \{(x_1, y_1), \dots, (x_n, y_n)\}$ — training set, where y_i is the maximum disparity for the picture x_i .

Find the algorithm $a : X \rightarrow Y$, which generalizes the target function.

If we replace y_i with the label of bin where y_i occurs we will get formulation of classification problem.

Classification statement has several advantages over regression:

- Classification is easy to train than regression,
- We can easily predict probability for each class (far, close, very far, very close, etc.).

So, further we consider the problem with 2 or 4 classes.

4 Training set preparation

Typical cityscapes data looks as follows.

```

In [8]: # good example nmb_image = 41, 12
        i = np.random.randint(0, len(disparity_paths))

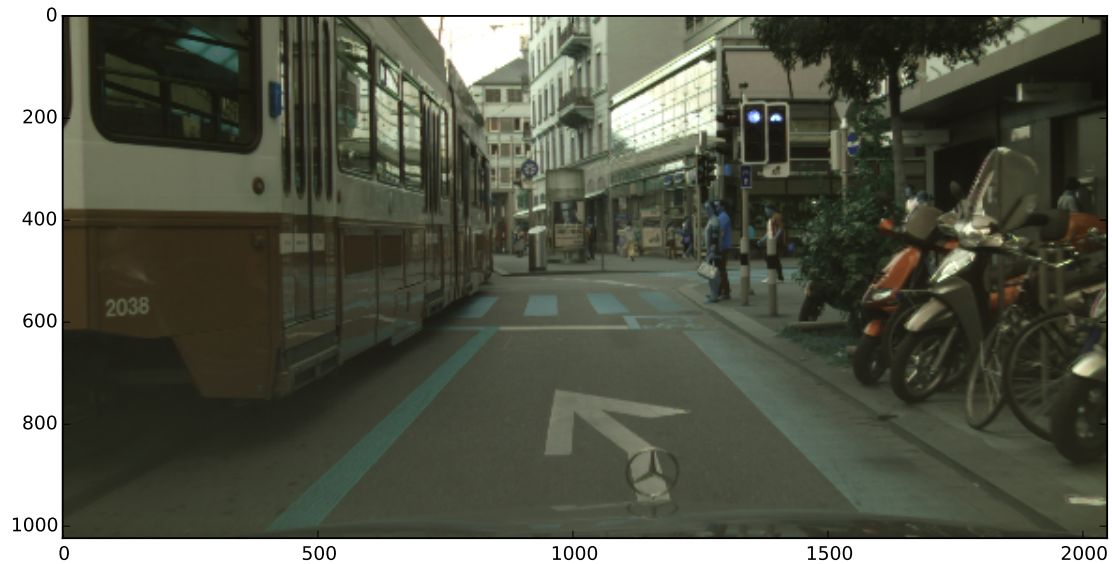
        disp = cv2.imread(disparity_paths[i])
        img = cv2.imread(imgs_paths[i])
        img_path = imgs_paths[i]
        anno_img = cv2.imread(anno_imgs_paths[i])

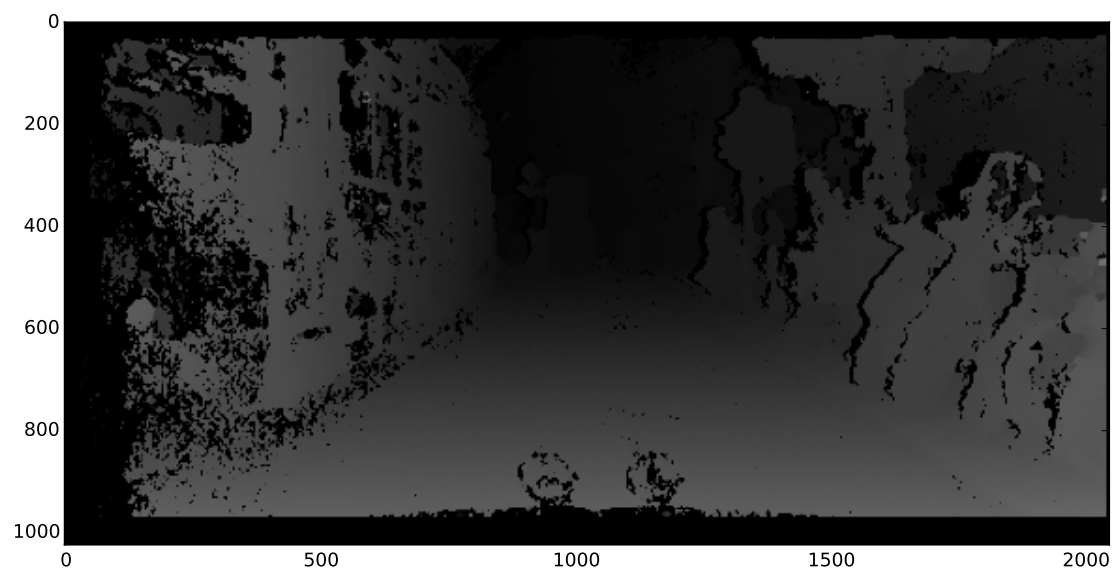
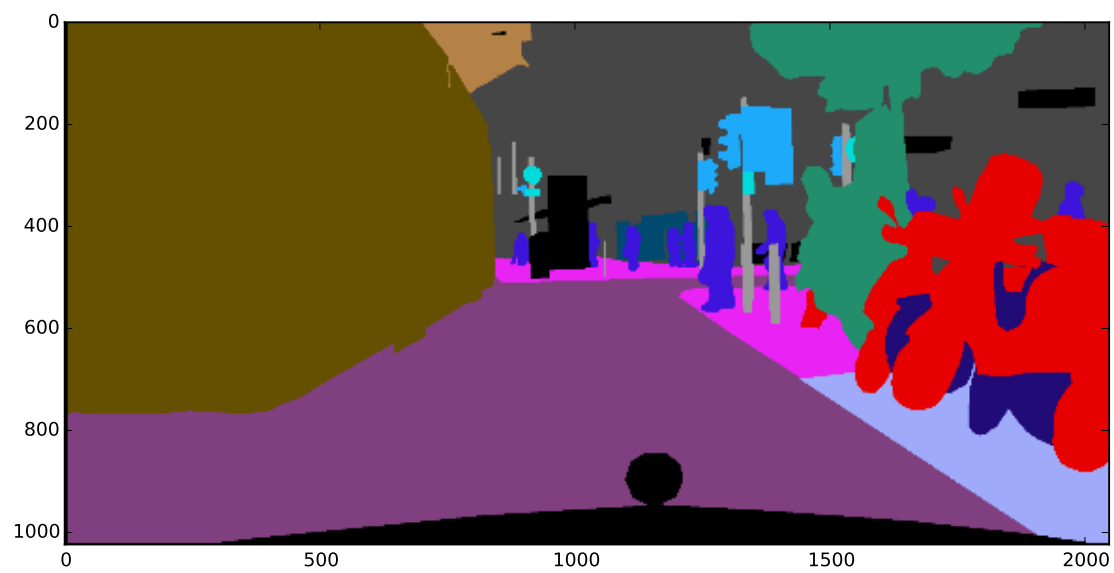
        Plot(img)
        Plot(anno_img)

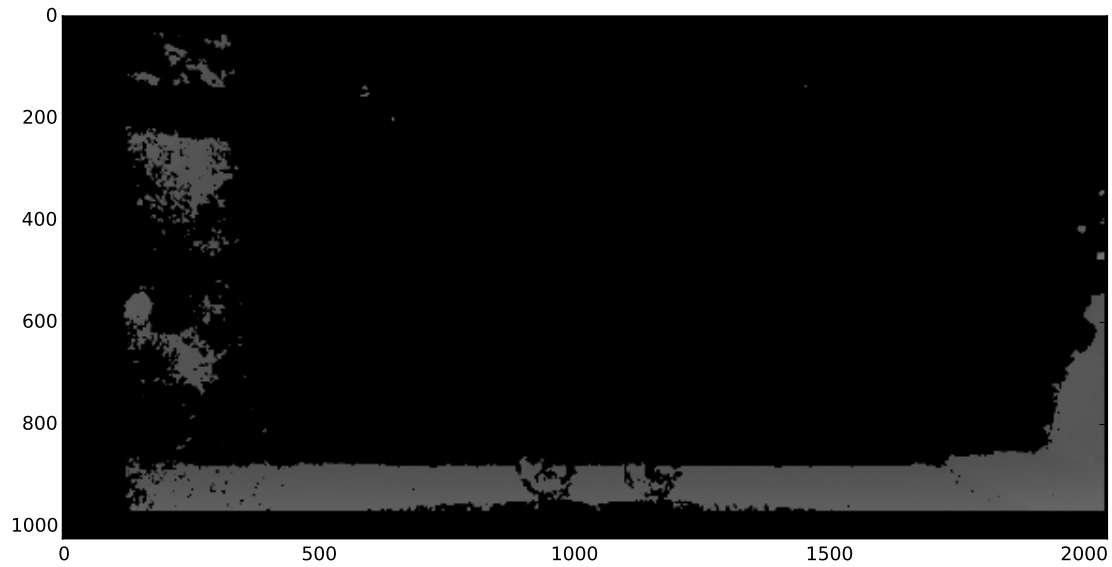
        ''' All disparity'''
        Plot(disp)

        ''' Disparity in percentile '''
        disp_perc = disp.copy()
        disp_perc[disp < np.percentile(disp, 90)] = 0
        Plot(disp_perc)

```

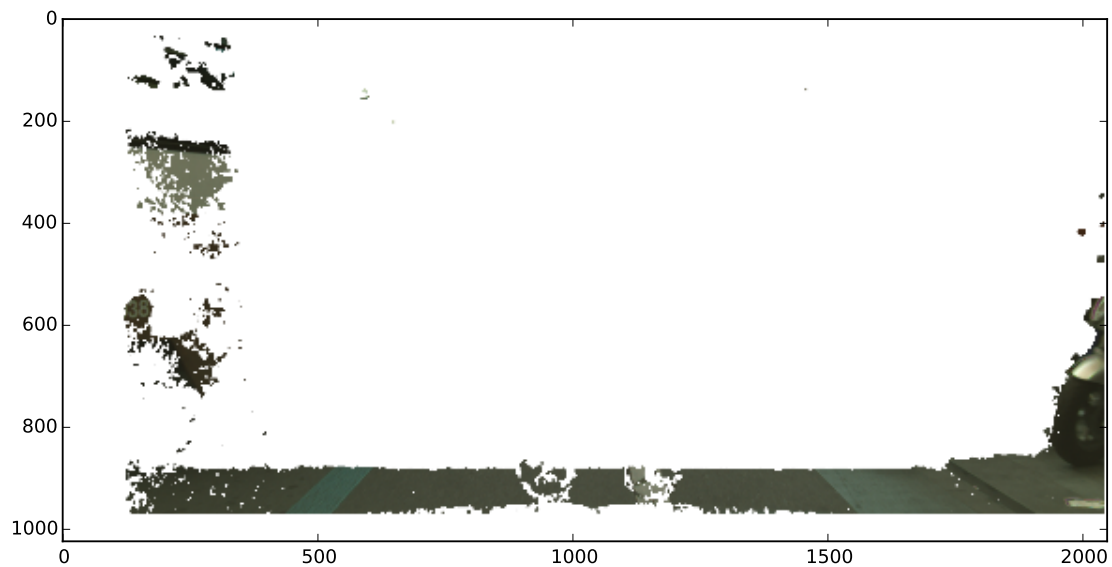






One can see that maximum disparity or any its percentile is meaningless for training set construction and physically trivial.

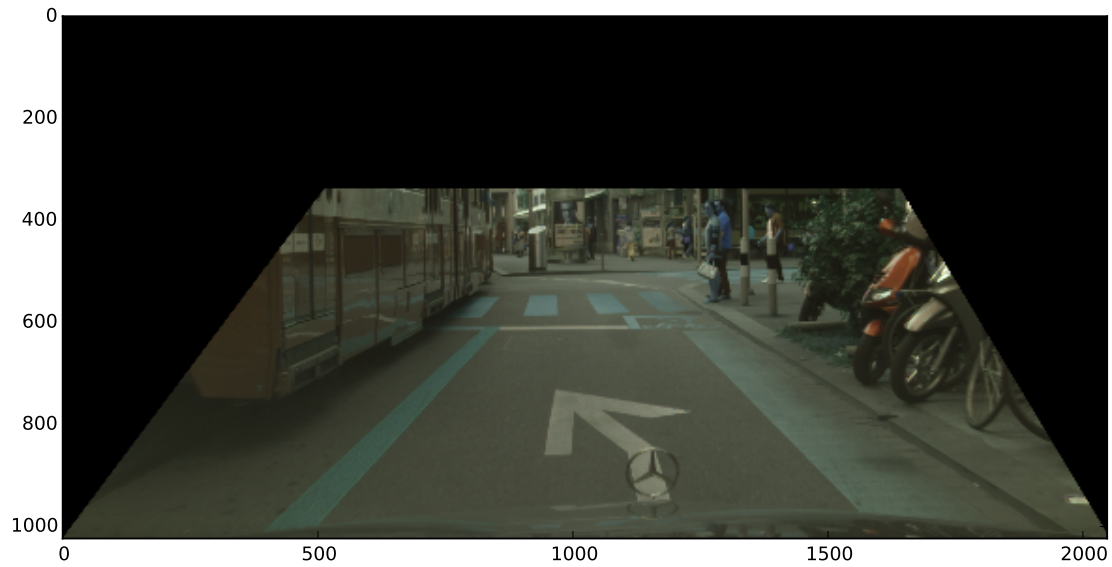
```
In [10]: img_perc = img.copy()
         img_perc[disp < np.percentile(disp, 90)] = 255
         Plot(img_perc)
```



So, first, the road obviously has to be removed and second, we extract target labels just for region of interest.

4.1 RoI selection

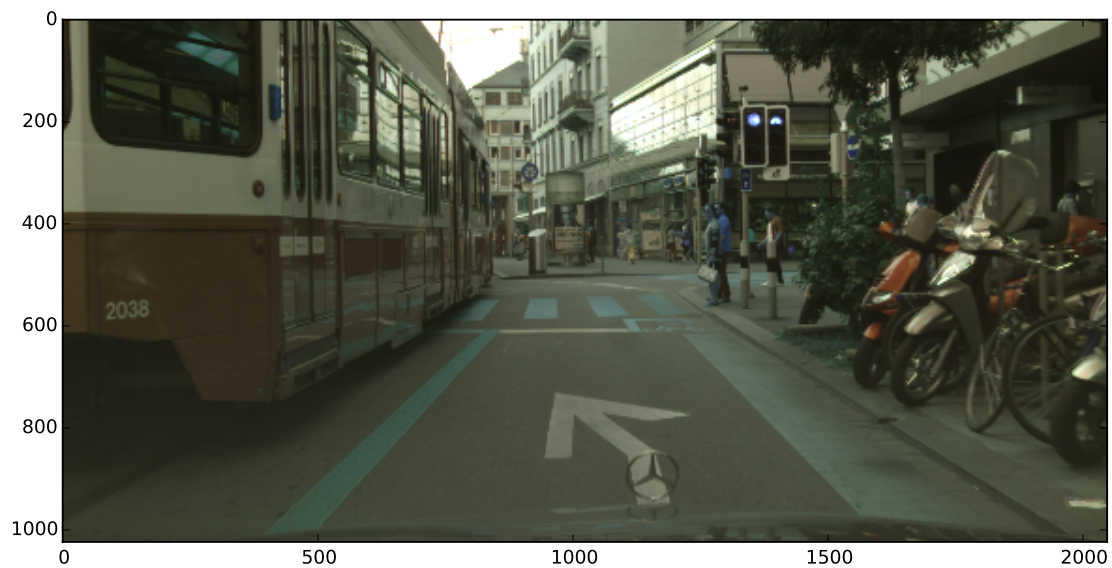
```
In [11]: Plot(CropRoI(cv2.imread(img_path, -1)))
```

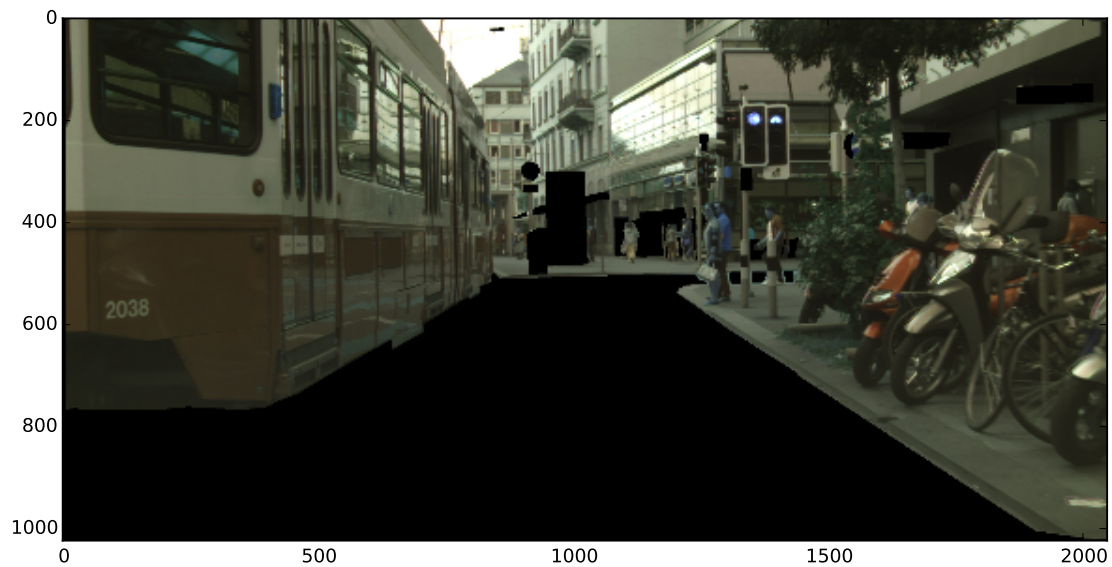
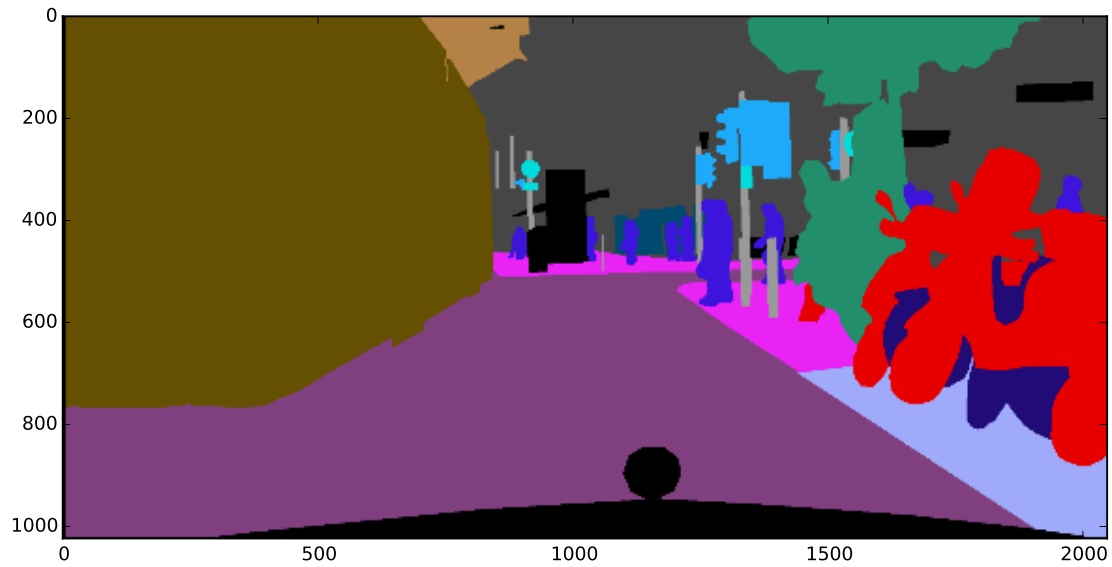


4.2 Exluding Road

We remove the road and car logo with the hood from the picture using fine segmentation provided by cityscapes team.

```
In [12]: Plot(img)
         Plot(anno_img)
         masked_img = ExcludeRoadDisp(img, anno_img, [(128, 64, 128), (0, 0, 0)])
         Plot(masked_img)
```

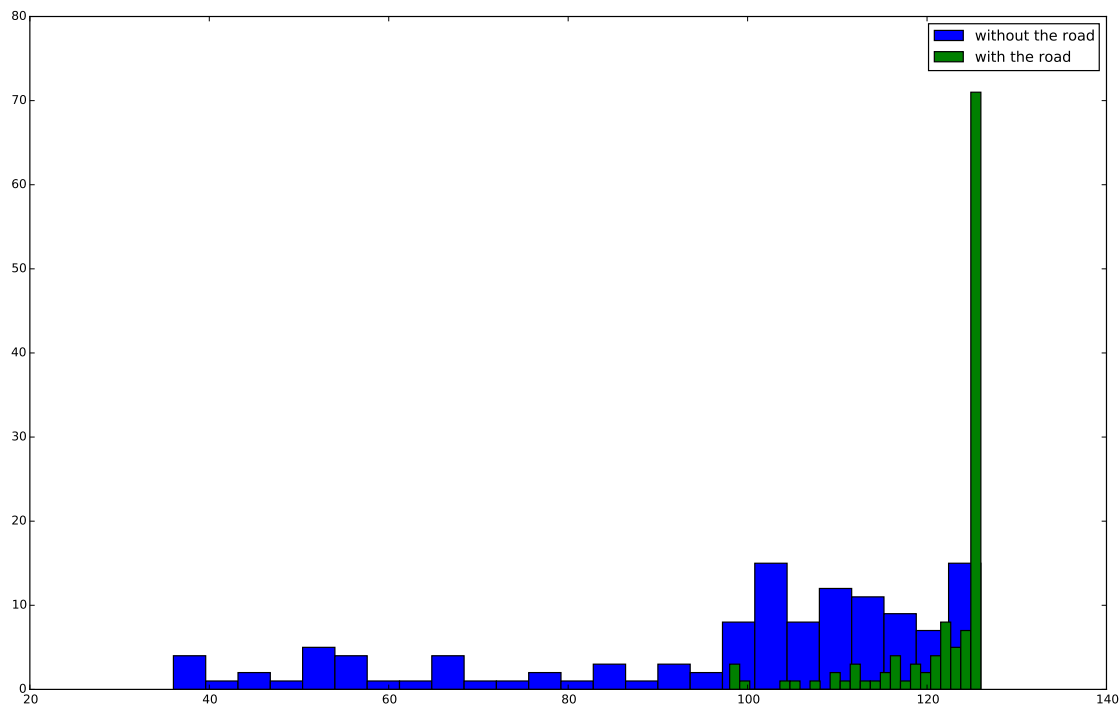




As a result, we got disparity distribution that makes sense and let the problem setting physically meaningful.

```
In [18]: plt.figure(figsize=(16, 10))
          plt.hist(max_disps_excl, bins=25, label='without the road')
          plt.hist(max_disps, bins=25, label='with the road')
          plt.legend(loc='best')
```

```
Out[18]: <matplotlib.legend.Legend at 0x4097fd0>
```



5 Features as CNN output

When solving classification problem for pictures it's important to choose suitable feature space F .

As soon as we determined feature space F and transformation $f : X \rightarrow F$, the problem we solve is simplified: find the algorithm $a : f(x) \mapsto y$, which generalizes the target function.

Advances in deep learning provide methods to solve the initial problem straightway, but we couldn't train the CNN due to lack of video card memory on our instance.

So, we decided to use spread pre-trained CNN output as that feature transform f .

We used VGG19 with reshaped input layer. Next we deleted everything after the convolutional layer and used that as features for the next processing. The structure of VGG19 that we used depicted further.



The result is a tensor of shape [512, 4, 9] raveled to the vector of length 18432.

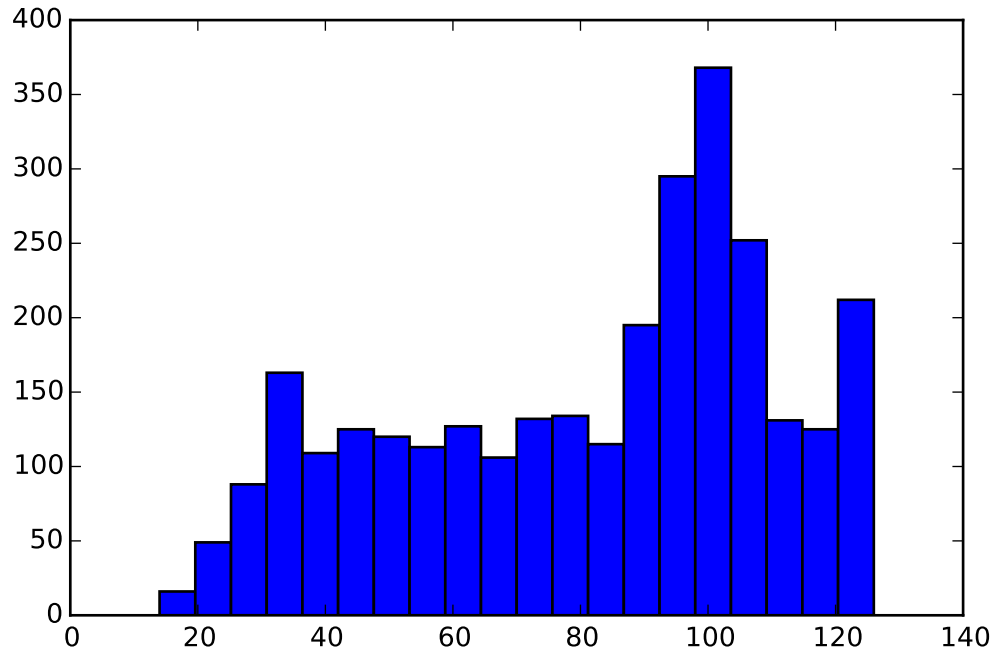
6 Classification

We applied different methods to solve classification problem.

Cityscapes training set has 2975 samples. It's possible to use extra train dataset of about 20000 samples (see below).

```
In [5]: plt.hist(y_train, bins=20)
```

```
Out[5]: (array([ 16.,  49.,  88., 163., 109., 125., 120., 113., 127.,
                106., 132., 134., 115., 195., 295., 368., 252., 131.,
                125., 212.]),
         array([ 14. ,  19.6,  25.2,  30.8,  36.4,  42. ,  47.6,  53.2,
                58.8,  64.4,  70. ,  75.6,  81.2,  86.8,  92.4,  98. ,
                103.6, 109.2, 114.8, 120.4, 126. ]),
         <a list of 20 Patch objects>)
```



```
In [6]: len(y_train), y_train
```

```
Out[6]: (2975, array([86, 93, 43, ..., 39, 97, 95], dtype=uint8))
```

```
In [129]: bins = [np.percentile(y_train, 0),
                  np.percentile(y_train, 25),
                  np.percentile(y_train, 50),
                  np.percentile(y_train, 75),
                  np.percentile(y_train, 100) + 1]
```

```
In [130]: bins
```

```
Out[130]: [14.0, 57.0, 89.0, 103.0, 127.0]
```

```
In [131]: binary_bins = [np.percentile(y_train, 0),
                         np.percentile(y_train, 50),
                         np.percentile(y_train, 100) + 1]
```

```
In [132]: binary_bins
```

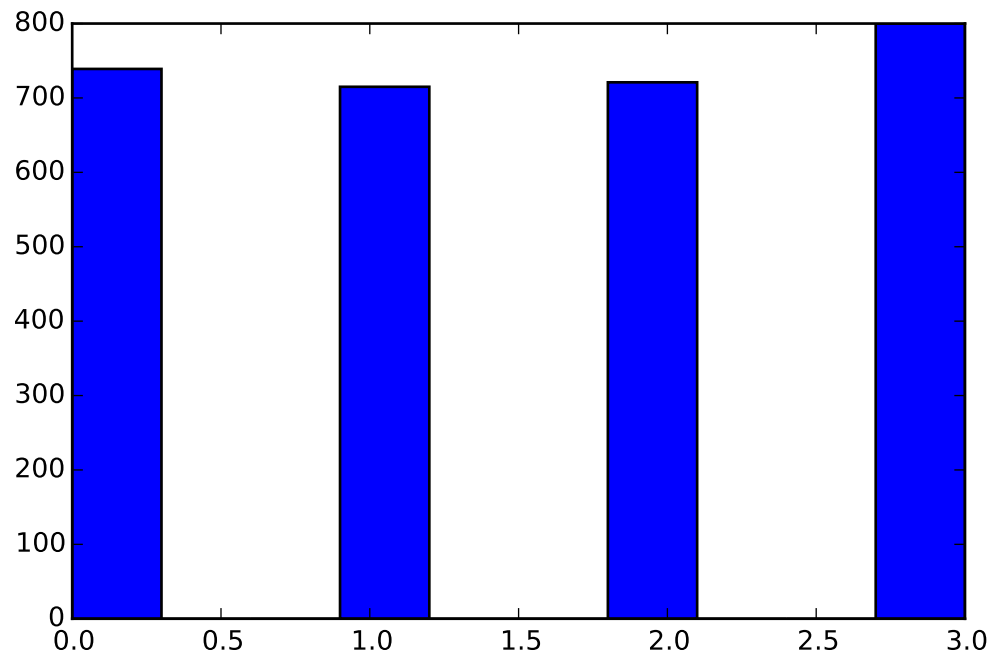
```
Out[132]: [14.0, 89.0, 127.0]
```

```
In [138]: y_train_binary = np.digitize(y_train, binary_bins) - 1
          y_val_binary = np.digitize(y_val, binary_bins) - 1
          y_test_binary = np.digitize(y_test, binary_bins) - 1
```

```
In [139]: y_train = np.digitize(y_train, bins) - 1
          y_val = np.digitize(y_val, bins) - 1
          y_test = np.digitize(y_test, bins) - 1
```

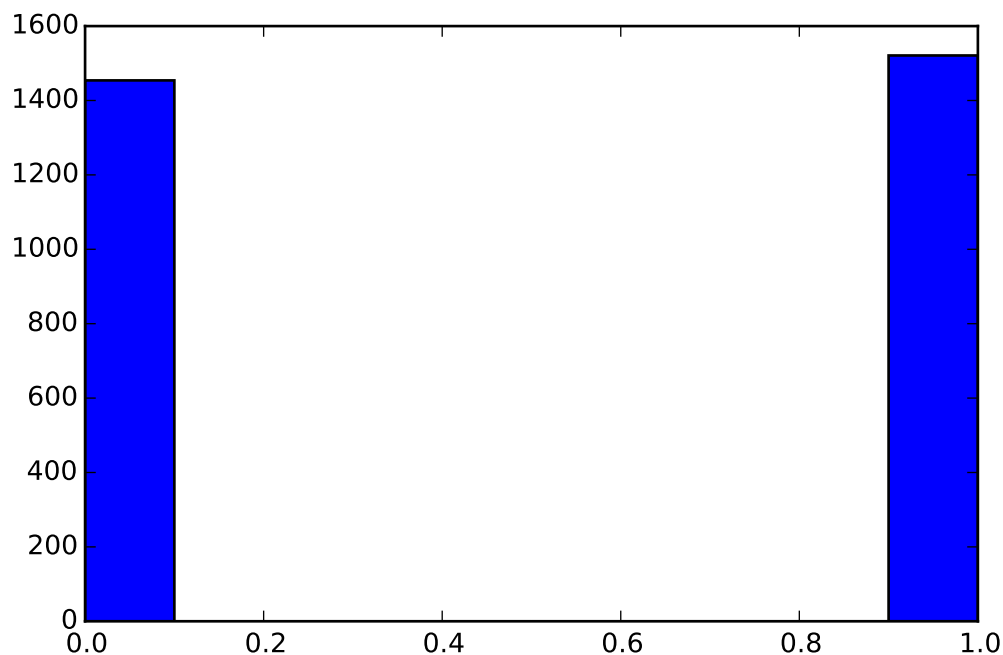
```
In [140]: plt.hist(y_train)
```

```
Out[140]: (array([ 739.,    0.,    0., 715.,    0.,    0., 721.,    0.,    0., 800.]),
          array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8,  2.1,  2.4,  2.7,  3. ]),
          <a list of 10 Patch objects>)
```



```
In [141]: plt.hist(y_train_binary)
```

```
Out[141]: (array([ 1454.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,
                  0., 1521.]),
          array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ]),
          <a list of 10 Patch objects>)
```



6.1 Binary classification

```
In [142]: clf = LogisticRegression(C=0.01, penalty='l1')
          clf.fit(f1_train, y_train_binary).score(f1_val, y_val_binary)
```

Out[142]: 0.878

Let's move on to multi-class.

6.2 Multi-class

6.2.1 Simple models

```
In [28]: clf = LogisticRegression(C=0.01, penalty='l1')
          clf.fit(f1_train, y_train).score(f1_val, y_val)
```

Out[28]: 0.68000000000000005

```
In [14]: from sklearn.preprocessing import Normalizer
```

```
normalizer = Normalizer().fit(f1_train)
f1_train_normalized = normalizer.transform(f1_train)
f1_val_normalized = normalizer.transform(f1_val)
```

```
In [66]: from sklearn.svm import LinearSVC
```

```
clf = LinearSVC(C=1, penalty='l2', loss='squared_hinge', dual=True, multi_class='crammer_singer')
clf.fit(f1_train_normalized, y_train).score(f1_val_normalized, y_val)
```

Out[66]: 0.67000000000000004

```
In [55]: clf = LogisticRegression(C=8, penalty='l1')
        clf.fit(f1_train_normalized, y_train).score(f1_val_normalized, y_val)
```

```
Out[55]: 0.67000000000000004
```

```
In [60]: from sklearn.svm import SVC
```

```
        clf = SVC(C=1, kernel='rbf', gamma=1)
        clf.fit(f1_train_normalized, y_train).score(f1_val_normalized, y_val)
```

```
Out[60]: 0.66800000000000004
```

```
In [61]: from sklearn.svm import SVC
```

```
        clf = SVC(C=1, kernel='poly', gamma=1, degree=2)
        clf.fit(f1_train_normalized, y_train).score(f1_val_normalized, y_val)
```

```
Out[61]: 0.67400000000000004
```

6.2.2 Boosting

```
In [46]: from sklearn.decomposition import PCA
```

```
        pca = PCA(n_components=1000).fit(f1_train_normalized)
```

```
        f1_train_pca = pca.transform(f1_train_normalized)
        f1_val_pca = pca.transform(f1_val_normalized)
```

```
In [45]: from sklearn.ensemble import GradientBoostingClassifier
```

```
        GradientBoostingClassifier(n_estimators=200).fit(f1_train_pca, y_train).score(f1_val_pca, y_val)
```

```
Out[45]: 0.59799999999999998
```

6.2.3 Stacking

```
In [76]: from stacking import Stacking
```

```
        basic_wildfowl = Stacking(base_estimators=[
            (LogisticRegression(C=0.01, penalty='l1').fit,
             lambda clf, X: clf.predict(X)),
            (lambda X, y: LinearSVC(C=1, penalty='l2', loss='squared_hinge',
                                    dual=True, multi_class='crammer_singer').fit(normalizer.transform(X)),
             lambda clf, X: clf.predict(normalizer.transform(X))),
            (lambda X, y: LogisticRegression(C=8, penalty='l1').fit(normalizer.transform(X), y),
             lambda clf, X: clf.predict(normalizer.transform(X))),
            (lambda X, y: SVC(C=1, kernel='rbf', gamma=1).fit(normalizer.transform(X), y),
             lambda clf, X: clf.predict(normalizer.transform(X))),
            (lambda X, y: SVC(C=1, kernel='poly', gamma=1, degree=2).fit(normalizer.transform(X), y),
             lambda clf, X: clf.predict(normalizer.transform(X)))],
            n_folds=5, extend_meta=False)
        basic_wildfowl.fit(np.array(f1_train), y_train)
```

```
/usr/lib64/python2.7/site-packages/sklearn/cross_validation.py:69: DeprecationWarning: The indices parameter
stacklevel=1)
```

```

Out[76]: Stacking(base_estimators=[(<bound method LogisticRegression.fit of LogisticRegression(C=0.01,
intercept_scaling=1, max_iter=100, multi_class='ovr',
penalty='l1', random_state=None, solver='liblinear', tol=0.0001,
verbose=0)>, <f...on <lambda> at 0x1d848848>), (<function <lambda> at 0x1d848758>, <
extend_meta=False, meta_fitter=None, n_folds=5)

In [77]: basic_wildfowl.fit_meta(SVC(C=5, kernel='poly', degree=2, gamma=1.).fit).score(np.array(f1_val), y_val)
Out[77]: 0.68999999999999995

In [80]: basic_wildfowl.fit_meta(SVC(C=1, kernel='poly', degree=2, gamma=1.).fit).score(np.array(f1_val), y_val)
Out[80]: 0.68999999999999995

In [81]: basic_wildfowl.fit_meta(SVC(C=10, kernel='poly', degree=2, gamma=1.).fit).score(np.array(f1_val), y_val)
Out[81]: 0.68999999999999995

In [82]: basic_wildfowl.fit_meta(SVC(C=1, kernel='rbf', gamma=1.).fit).score(np.array(f1_val), y_val)
Out[82]: 0.69799999999999995

In [100]: basic_wildfowl.fit_meta(SVC(C=1, kernel='rbf', gamma=0.5).fit).score(np.array(f1_val), y_val)
Out[100]: 0.70399999999999996

In [109]: basic_wildfowl.fit_meta(SVC(C=1, kernel='rbf', gamma=0.3).fit).score(np.array(f1_val), y_val)
Out[109]: 0.70199999999999996

In [143]: basic_wildfowl.fit_meta(SVC(C=1, kernel='rbf', gamma=0.2).fit).score(np.array(f1_val), y_val)
Out[143]: 0.70599999999999996

In [105]: basic_wildfowl.fit_meta(SVC(C=1, kernel='rbf', gamma=0.1).fit).score(np.array(f1_val), y_val)
Out[105]: 0.70399999999999996

In [83]: basic_wildfowl.fit_meta(SVC(C=5, kernel='rbf', gamma=1.).fit).score(np.array(f1_val), y_val)
Out[83]: 0.69599999999999995

In [84]: basic_wildfowl.fit_meta(SVC(C=0.5, kernel='rbf', gamma=1.).fit).score(np.array(f1_val), y_val)
Out[84]: 0.69399999999999995

In [85]: basic_wildfowl.fit_meta(LogisticRegression(C=1, penalty='l1').fit).score(np.array(f1_val), y_val)
Out[85]: 0.67000000000000004

In [89]: basic_wildfowl.fit_meta(LogisticRegression(C=10, penalty='l2').fit).score(np.array(f1_val), y_val)
Out[89]: 0.67000000000000004

In [92]: basic_wildfowl.fit_meta(RandomForestClassifier(n_estimators=100).fit).score(np.array(f1_val), y_val)
Out[92]: 0.69599999999999995

In [94]: basic_wildfowl.fit_meta(RandomForestClassifier(n_estimators=50, max_depth=10).fit).score(np.array(f1_val), y_val)
Out[94]: 0.68999999999999995

```

```

In [95]: basic_wildfowl.fit_meta(RandomForestClassifier(n_estimators=50, max_depth=20).fit).score(np.ar
Out[95]: 0.6959999999999995

In [78]: basic_wildfowl.fit_meta(GradientBoostingClassifier(n_estimators=200).fit).score(np.array(f1_val)
Out[78]: 0.6939999999999995

In [ ]: from sklearn.ensemble import AdaBoostClassifier

In [101]: basic_wildfowl.fit_meta(AdaBoostClassifier(n_estimators=30).fit).score(np.array(f1_val), y_val)
Out[101]: 0.67000000000000004

In [96]: basic_wildfowl.fit_meta(AdaBoostClassifier(n_estimators=50).fit).score(np.array(f1_val), y_val)
Out[96]: 0.6879999999999994

In [97]: basic_wildfowl.fit_meta(AdaBoostClassifier(n_estimators=100).fit).score(np.array(f1_val), y_val)
Out[97]: 0.68000000000000005

```

6.2.4 With the road + segmentation

```

In [65]: clf = LogisticRegression(C=0.005, penalty='l1')
         clf.fit(np.hstack((np.array(f1_train), np.array(f2_train))), y_train).score(np.hstack((np.array
Out[65]: 0.67600000000000005

```

6.2.5 Road deleted + segmentation

```

In [103]: clf = LogisticRegression(C=0.001, penalty='l1')
         clf.fit(np.hstack((np.array(f1_train), np.array(f2_train))), y_train).score(np.hstack((np.array
Out[103]: 0.66600000000000004

```

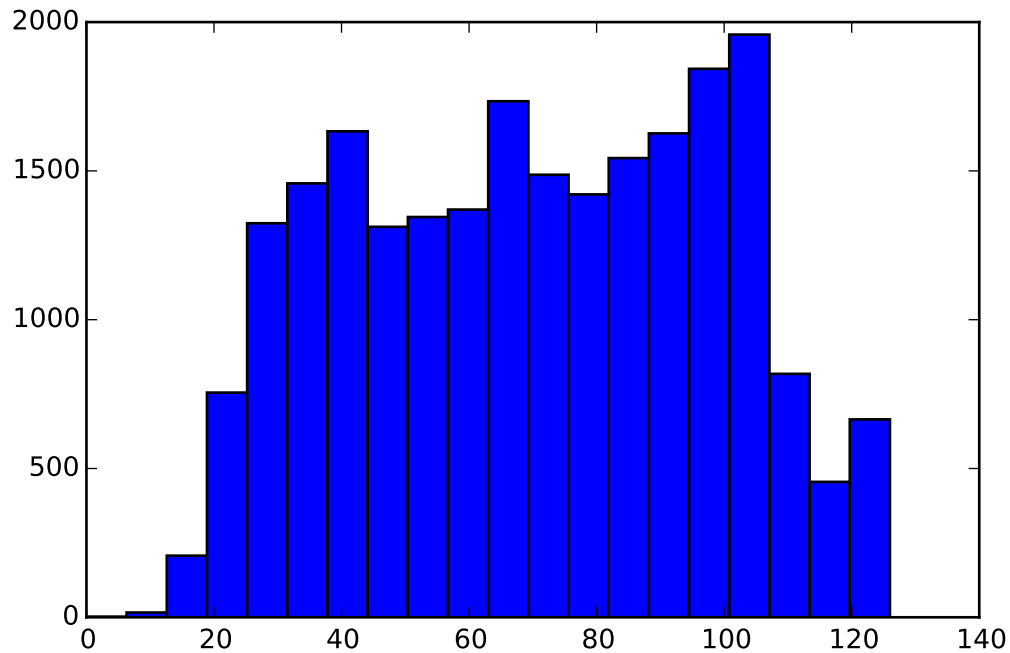
6.2.6 Extra train dataset

```

In [161]: plt.hist(y_train, bins=20)

Out[161]: (array([  2.,  16., 207., 755., 1324., 1458., 1633., 1312.,
        1345., 1370., 1734., 1487., 1421., 1543., 1626., 1843.,
        1958.,  818.,  455.,  665.]),
         array([  0.,   6.3,  12.6,  18.9,  25.2,  31.5,  37.8,  44.1,
        50.4,  56.7,  63. ,  69.3,  75.6,  81.9,  88.2,  94.5,
        100.8, 107.1, 113.4, 119.7, 126. ]),
         <a list of 20 Patch objects>)

```



```
In [162]: bins_extra = [np.percentile(y_train, 0),
                        np.percentile(y_train, 25),
                        np.percentile(y_train, 50),
                        np.percentile(y_train, 75),
                        np.percentile(y_train, 100) + 1]
```

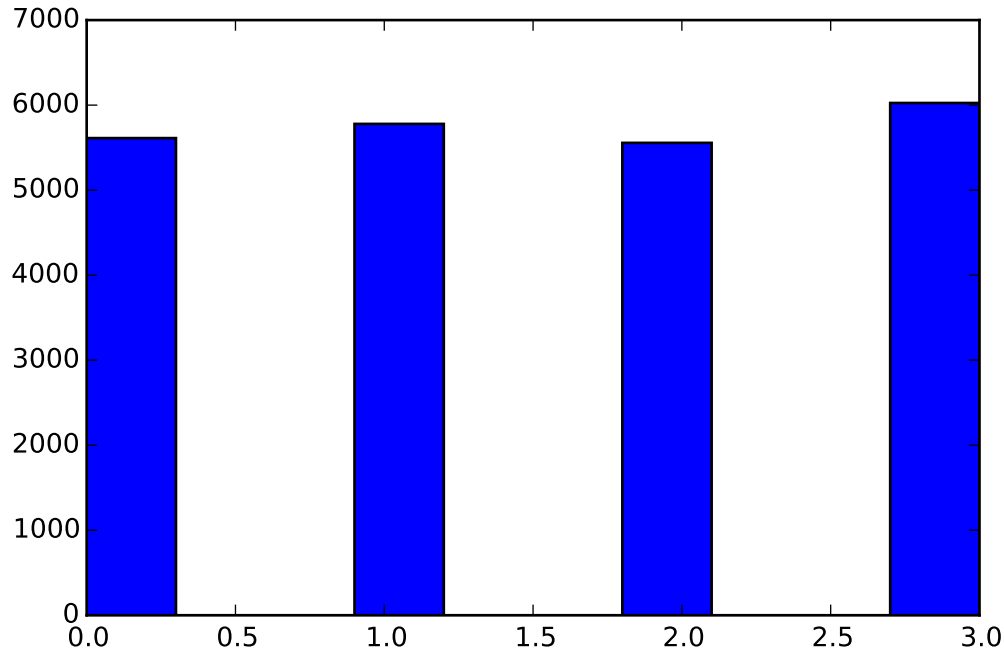
```
In [163]: bins_extra
```

```
Out[163]: [0.0, 46.0, 71.0, 94.0, 127.0]
```

```
In [164]: y_train = np.digitize(y_train, bins_extra) - 1
          y_val = np.digitize(y_val, bins_extra) - 1
          y_test = np.digitize(y_test, bins_extra) - 1
```

```
In [165]: plt.hist(y_train)
```

```
Out[165]: (array([ 5612.,    0.,    0.,  5779.,    0.,    0.,  5556.,    0.,
                   0.,  6025.]),
          array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8,  2.1,  2.4,  2.7,  3. ]),
          <a list of 10 Patch objects>)
```

```
In [ ]: clf = LogisticRegression(C=0.01, penalty='l1')
        clf.fit(f1_train, y_train).score(f1_val, y_val)
```

```
Out[ ]: 0.70199999999999996
```

One can see that using extra train dataset doesn't increase result significantly. It may be due to the fact that we didn't have fine segmentation for extra train samples. Instead we had to use coarse segmentation of poor quality.

6.3 ANN

We tried the structure that follows the structure of VGG19. The difference was to add elu nonlinearities to all the dense layers. The structure is the following:

- DenseLayer - 2048 units + elu
- DropoutLayer - 50%
- DenseLayer - 2048 units + elu
- DropoutLayer - 50%
- DenseLayer - 1024 units + elu
- DropoutLayer - 50%
- DenseLayer - 512 units + elu
- DropoutLayer - 50%
- DenseLayer - 256 units + elu
- DropoutLayer - 50%
- DenseLayer - Softmax

First we tried to add these layers to output of VGG19 features leayers. The problem was that it required much higher amount of memory and we couldn't use batches even of size 50. Next we used the described network as a separate classifier. It allowed us to use much higher batch sizes.

Furthermore we faced the problem of overfitting. In order to deal with that we introduced l2 and l1 regularizations to the loss.

Talking about loss it is important to mention that we decided to use categorical_crossentropy as a loss function. It was motivated by the fact that we dealt with different sizes of classes (we started with binary classification problem and went to 4 classes classification).

We had to play with regularization weights. Increasing and decreasing it we could limit overfitting and improve the loss on validation set.

6.3.1 Network deleted road, classes

```
In [37]: with open('data_rem_road.pkl', 'rb') as f:
        X_train, Seg_train, y_train, X_val, Seg_val, y_val, X_test, Seg_test, y_test = pickle.load()

        with open('features_rem_road.pkl', 'rb') as f:
            f1_train, f1_val, f1_test, f2_train, f2_val, f2_test = pickle.load(f)

        bins = [np.percentile(y_train, 0),
                 np.percentile(y_train, 25),
                 np.percentile(y_train, 50),
                 np.percentile(y_train, 75),
                 np.percentile(y_train, 100) + 1]

        y_train = np.digitize(y_train, bins) - 1
        y_val = np.digitize(y_val, bins) - 1
        y_test = np.digitize(y_test, bins) - 1

        input_layer = InputLayer((None, f1_train[0].shape[0]))
        l11 = DenseLayer(input_layer, num_units=2048, nonlinearity=lasagne.nonlinearities.elu)
        dr1 = DropoutLayer(l11, p=0.5)
        l12 = DenseLayer(dr1, num_units=2048, nonlinearity=lasagne.nonlinearities.elu)
        dr3 = DropoutLayer(l12, p=0.5)
        l13 = DenseLayer(dr3, num_units=1024, nonlinearity=lasagne.nonlinearities.elu)
        dr4 = DropoutLayer(l13, p=0.5)
        l14 = DenseLayer(dr4, num_units=512, nonlinearity=lasagne.nonlinearities.elu)
        dr5 = DropoutLayer(l14, p=0.5)
        l15 = DenseLayer(dr5, num_units=256, nonlinearity=lasagne.nonlinearities.elu)
        dr6 = DropoutLayer(l15, p=0.5)
        percentile = DenseLayer(dr6, num_units=len(set(y_train)), nonlinearity=softmax)

        target_y = T.vector("target Y integer", dtype='int32')
        input_X = T.matrix('X')

        probs = lasagne.layers.get_output(percentile, input_X, deterministic = True)
        y_predicted = np.argmax(probs, axis=1)

        #all network weights (shared variables)
        all_weights = lasagne.layers.get_all_params(percentile)
        print(all_weights)

        #Mean categorical crossentropy as a loss function - similar to logistic loss but for multiclass
        loss = lasagne.objectives.categorical_crossentropy(probs, target_y).mean()
        l2_penalty = regularize_layer_params(percentile, l2) * 1e-1
        l1_penalty = regularize_layer_params(percentile, l1) * 1e-3
        loss += l2_penalty + l1_penalty

        #prediction accuracy
```

```

accuracy = lasagne.objectives.categorical_accuracy(probs, target_y).mean()

#This function computes gradient AND composes weight updates just like you did earlier
updates_sgd = lasagne.updates.adagrad(loss, all_weights, learning_rate=0.01)

#function that computes loss and updates weights
train_fun = theano.function([input_X, target_y], [loss, accuracy], updates = updates_sgd)

#function that just computes accuracy
accuracy_fun = theano.function([input_X, target_y], accuracy)

num_epochs = 30 #amount of passes through the data

batch_size = 100 #number of samples processed at each function call

for epoch in range(num_epochs):
    # In each epoch, we do a full pass over the training data:
    train_err = 0
    train_acc = 0
    train_batches = 0
    start_time = time.time()
    for batch in iterate_minibatches(f1_train, y_train, batch_size):
        inputs, targets = batch
        train_err_batch, train_acc_batch = train_fun(inputs, targets.astype(np.int32))
        train_err += train_err_batch
        train_acc += train_acc_batch
        train_batches += 1

    # And a full pass over the validation data:
    val_acc = 0
    val_batches = 0
    for batch in iterate_minibatches(f1_val, y_val, batch_size):
        inputs, targets = batch
        val_acc += accuracy_fun(inputs, targets.astype(np.int32))
        val_batches += 1

    # Then we print the results for this epoch:
    print("Epoch {} of {} took {:.3f}s".format(
        epoch + 1, num_epochs, time.time() - start_time))

    print("  training loss (in-iteration):\t\t{:.6f}".format(train_err / train_batches))
    print("  train accuracy:\t\t{:.2f} %".format(
        train_acc / train_batches * 100))
    print("  validation accuracy:\t\t{:.2f} %".format(
        val_acc / val_batches * 100))

[W, b, W, b, W, b, W, b, W, b, W, b]
Epoch 1 of 30 took 1.998s
  training loss (in-iteration):          3749.619600
  train accuracy:          28.59 %
  validation accuracy:          26.00 %
Epoch 2 of 30 took 2.015s
  training loss (in-iteration):          3.113019

```

train accuracy:	32.24 %	
validation accuracy:	46.60 %	
Epoch 3 of 30 took 1.982s		
training loss (in-iteration):		2.176733
train accuracy:	45.90 %	
validation accuracy:	49.40 %	
Epoch 4 of 30 took 2.010s		
training loss (in-iteration):		1.896570
train accuracy:	52.72 %	
validation accuracy:	58.00 %	
Epoch 5 of 30 took 1.978s		
training loss (in-iteration):		1.765641
train accuracy:	58.38 %	
validation accuracy:	52.20 %	
Epoch 6 of 30 took 2.000s		
training loss (in-iteration):		1.671071
train accuracy:	63.17 %	
validation accuracy:	62.00 %	
Epoch 7 of 30 took 2.000s		
training loss (in-iteration):		1.551463
train accuracy:	69.31 %	
validation accuracy:	55.20 %	
Epoch 8 of 30 took 2.111s		
training loss (in-iteration):		1.477413
train accuracy:	71.52 %	
validation accuracy:	66.60 %	
Epoch 9 of 30 took 2.005s		
training loss (in-iteration):		1.404215
train accuracy:	76.62 %	
validation accuracy:	59.40 %	
Epoch 10 of 30 took 1.980s		
training loss (in-iteration):		1.373313
train accuracy:	76.55 %	
validation accuracy:	67.40 %	
Epoch 11 of 30 took 1.987s		
training loss (in-iteration):		1.343516
train accuracy:	77.24 %	
validation accuracy:	60.80 %	
Epoch 12 of 30 took 2.008s		
training loss (in-iteration):		1.256678
train accuracy:	81.34 %	
validation accuracy:	61.20 %	
Epoch 13 of 30 took 2.102s		
training loss (in-iteration):		1.214769
train accuracy:	83.93 %	
validation accuracy:	66.20 %	
Epoch 14 of 30 took 1.999s		
training loss (in-iteration):		1.203365
train accuracy:	84.66 %	
validation accuracy:	61.60 %	
Epoch 15 of 30 took 1.974s		
training loss (in-iteration):		1.129455
train accuracy:	87.41 %	
validation accuracy:	67.00 %	

```

Epoch 16 of 30 took 2.126s
  training loss (in-iteration):      1.263559
  train accuracy:                   85.00 %
  validation accuracy:               67.40 %
Epoch 17 of 30 took 1.980s
  training loss (in-iteration):      1.062010
  train accuracy:                   91.52 %
  validation accuracy:               65.40 %
Epoch 18 of 30 took 1.974s
  training loss (in-iteration):      1.000695
  train accuracy:                   93.72 %
  validation accuracy:               64.40 %
Epoch 19 of 30 took 2.005s
  training loss (in-iteration):      0.974304
  train accuracy:                   94.59 %
  validation accuracy:               64.00 %
Epoch 20 of 30 took 1.980s
  training loss (in-iteration):      0.925185
  train accuracy:                   96.62 %
  validation accuracy:               62.00 %
Epoch 21 of 30 took 2.003s
  training loss (in-iteration):      0.937583
  train accuracy:                   96.28 %
  validation accuracy:               62.60 %
Epoch 22 of 30 took 1.997s
  training loss (in-iteration):      1.025416
  train accuracy:                   93.21 %
  validation accuracy:               62.00 %
Epoch 23 of 30 took 1.987s
  training loss (in-iteration):      1.056850
  train accuracy:                   92.55 %
  validation accuracy:               63.80 %
Epoch 24 of 30 took 1.997s
  training loss (in-iteration):      0.880320
  train accuracy:                   98.07 %
  validation accuracy:               64.40 %
Epoch 25 of 30 took 2.182s
  training loss (in-iteration):      0.861131
  train accuracy:                   98.76 %
  validation accuracy:               64.80 %
Epoch 26 of 30 took 2.169s
  training loss (in-iteration):      0.866686
  train accuracy:                   98.07 %
  validation accuracy:               64.60 %
Epoch 27 of 30 took 2.097s
  training loss (in-iteration):      0.839886
  train accuracy:                   99.03 %
  validation accuracy:               64.80 %

```

KeyboardInterrupt

Traceback (most recent call last)

```

<ipython-input-37-d82e6a4058e4> in <module>()
    68     for batch in iterate_minibatches(f1_train, y_train, batch_size):
    69         inputs, targets = batch
---> 70         train_err_batch, train_acc_batch = train_fun(inputs, targets.astype(np.int32))
    71         train_err += train_err_batch
    72         train_acc += train_acc_batch

/usr/lib/python2.7/site-packages/theano/compile/function_module.py in __call__(self, *args, **kwargs)
    813         s.storage[0] = s.type.filter(
    814             arg, strict=s.strict,
--> 815             allow_downcast=s.allow_downcast)
    816
    817         except Exception as e:

/usr/lib/python2.7/site-packages/theano/tensor/type.py in filter(self, data, strict, allow_downcast)
    148         # data has to be converted.
    149         # Check that this conversion is lossless
--> 150         converted_data = theano._asarray(data, self.dtype)
    151         # We use the 'values_eq' static function from TensorType
    152         # to handle NaN values.

/usr/lib/python2.7/site-packages/theano/misc/safe_asarray.py in _asarray(a, dtype, order)
    32         dtype = theano.config.floatX
    33         dtype = numpy.dtype(dtype) # Convert into dtype object.
---> 34         rval = numpy.asarray(a, dtype=dtype, order=order)
    35         # Note that dtype comparison must be done by comparing their 'num'
    36         # attribute. One cannot assume that two identical data types are pointers

/usr/lib64/python2.7/site-packages/numpy/core/numeric.py in asarray(a, dtype, order)
    480
    481     """
--> 482     return array(a, dtype, copy=False, order=order)
    483
    484 def asanyarray(a, dtype=None, order=None):

```

KeyboardInterrupt:

6.3.2 Network with road, 4 classes

```

In [ ]: with open('data_basic.pkl', 'rb') as f:
        X_train, Seg_train, y_train, X_val, Seg_val, y_val, X_test, Seg_test, y_test = pickle.load(f)

        with open('features_basic.pkl', 'rb') as f:
            f1_train, f1_val, f1_test, f2_train, f2_val, f2_test = pickle.load(f)

        bins = [np.percentile(y_train, 0),
                 np.percentile(y_train, 25),
                 np.percentile(y_train, 50),

```

```

        np.percentile(y_train, 75),
        np.percentile(y_train, 100) + 1]

y_train = np.digitize(y_train, bins) - 1
y_val = np.digitize(y_val, bins) - 1
y_test = np.digitize(y_test, bins) - 1

In [36]: input_layer = InputLayer((None, f1_train[0].shape[0]))
        l11 = DenseLayer(input_layer, num_units=2048, nonlinearity=lasagne.nonlinearities.elu)
        dr1 = DropoutLayer(l11, p=0.5)
        l12 = DenseLayer(dr1, num_units=2048, nonlinearity=lasagne.nonlinearities.elu)
        dr3 = DropoutLayer(l12, p=0.5)
        l13 = DenseLayer(dr3, num_units=1024, nonlinearity=lasagne.nonlinearities.elu)
        dr4 = DropoutLayer(l13, p=0.5)
        l14 = DenseLayer(dr4, num_units=512, nonlinearity=lasagne.nonlinearities.elu)
        dr5 = DropoutLayer(l14, p=0.5)
        l15 = DenseLayer(dr5, num_units=256, nonlinearity=lasagne.nonlinearities.elu)
        dr6 = DropoutLayer(l15, p=0.5)
        percentile = DenseLayer(dr6, num_units=len(set(y_train)), nonlinearity=softmax)

        target_y = T.vector("target Y integer", dtype='int32')
        input_X = T.matrix('X')

        probs = lasagne.layers.get_output(percentile, input_X, deterministic = True)
        y_predicted = np.argmax(probs, axis=1)

        #all network weights (shared variables)
        all_weights = lasagne.layers.get_all_params(percentile)
        print(all_weights)

        #Mean categorical crossentropy as a loss function - similar to logistic loss but for multiclass
        loss = lasagne.objectives.categorical_crossentropy(probs, target_y).mean()
        l2_penalty = regularize_layer_params(percentile, l2) * 1e-1
        l1_penalty = regularize_layer_params(percentile, l1) * 1e-2
        loss += l2_penalty + l1_penalty

        #prediction accuracy
        accuracy = lasagne.objectives.categorical_accuracy(probs, target_y).mean()

        #This function computes gradient AND composes weight updates just like you did earlier
        updates_sgd = lasagne.updates.adagrad(loss, all_weights, learning_rate=0.005)

        #function that computes loss and updates weights
        train_fun = theano.function([input_X, target_y], [loss, accuracy], updates = updates_sgd)

        #function that just computes accuracy
        accuracy_fun = theano.function([input_X, target_y], accuracy)

        num_epochs = 300 #amount of passes through the data

        batch_size = 100 #number of samples processed at each function call

        for epoch in range(num_epochs):
            # In each epoch, we do a full pass over the training data:

```

```

train_err = 0
train_acc = 0
train_batches = 0
start_time = time.time()
for batch in iterate_minibatches(f1_train, y_train, batch_size):
    inputs, targets = batch
    train_err_batch, train_acc_batch = train_fun(inputs, targets.astype(np.int32))
    train_err += train_err_batch
    train_acc += train_acc_batch
    train_batches += 1

# And a full pass over the validation data:
val_acc = 0
val_batches = 0
for batch in iterate_minibatches(f1_val, y_val, batch_size):
    inputs, targets = batch
    val_acc += accuracy_fun(inputs, targets.astype(np.int32))
    val_batches += 1

# Then we print the results for this epoch:
print("Epoch {} of {} took {:.3f}s".format(
    epoch + 1, num_epochs, time.time() - start_time))

print("  training loss (in-iteration):\t\t{:.6f}".format(train_err / train_batches))
print("  train accuracy:\t\t{:.2f} %".format(
    train_acc / train_batches * 100))
print("  validation accuracy:\t\t{:.2f} %".format(
    val_acc / val_batches * 100))

```

```

[W, b, W, b, W, b, W, b, W, b, W, b]
Epoch 1 of 300 took 2.036s
  training loss (in-iteration):          140.387504
  train accuracy:          31.28 %
  validation accuracy:          52.80 %
Epoch 2 of 300 took 2.016s
  training loss (in-iteration):          13.295092
  train accuracy:          55.17 %
  validation accuracy:          58.20 %
Epoch 3 of 300 took 2.002s
  training loss (in-iteration):          12.705572
  train accuracy:          62.45 %
  validation accuracy:          60.40 %
Epoch 4 of 300 took 2.003s
  training loss (in-iteration):          12.226186
  train accuracy:          68.86 %
  validation accuracy:          60.40 %
Epoch 5 of 300 took 2.002s
  training loss (in-iteration):          11.880651
  train accuracy:          71.07 %
  validation accuracy:          60.20 %
Epoch 6 of 300 took 2.014s
  training loss (in-iteration):          11.404465
  train accuracy:          79.07 %

```


validation accuracy:	61.80 %	
Epoch 7 of 300 took 2.018s		
training loss (in-iteration):		11.053822
train accuracy:	84.86 %	
validation accuracy:	65.00 %	
Epoch 8 of 300 took 1.896s		
training loss (in-iteration):		10.977491
train accuracy:	79.59 %	
validation accuracy:	61.00 %	
Epoch 9 of 300 took 2.035s		
training loss (in-iteration):		10.554585
train accuracy:	87.14 %	
validation accuracy:	65.00 %	
Epoch 10 of 300 took 1.963s		
training loss (in-iteration):		10.316450
train accuracy:	90.83 %	
validation accuracy:	64.80 %	
Epoch 11 of 300 took 1.936s		
training loss (in-iteration):		10.013761
train accuracy:	95.55 %	
validation accuracy:	63.40 %	
Epoch 12 of 300 took 1.933s		
training loss (in-iteration):		9.810423
train accuracy:	97.24 %	
validation accuracy:	63.80 %	
Epoch 13 of 300 took 1.933s		
training loss (in-iteration):		9.722251
train accuracy:	95.52 %	
validation accuracy:	63.00 %	
Epoch 14 of 300 took 1.922s		
training loss (in-iteration):		9.467655
train accuracy:	99.24 %	
validation accuracy:	64.40 %	
Epoch 15 of 300 took 1.951s		
training loss (in-iteration):		9.321485
train accuracy:	99.38 %	
validation accuracy:	66.20 %	
Epoch 16 of 300 took 1.924s		
training loss (in-iteration):		9.184269
train accuracy:	99.72 %	
validation accuracy:	65.60 %	
Epoch 17 of 300 took 1.955s		
training loss (in-iteration):		9.060074
train accuracy:	99.86 %	
validation accuracy:	65.60 %	
Epoch 18 of 300 took 2.053s		
training loss (in-iteration):		8.943007
train accuracy:	99.93 %	
validation accuracy:	64.40 %	

KeyboardInterrupt

Traceback (most recent call last)

```

<ipython-input-36-c68afadbdaafd> in <module>()
    52     for batch in iterate_minibatches(f1_train, y_train, batch_size):
    53         inputs, targets = batch
---> 54         train_err_batch, train_acc_batch = train_fun(inputs, targets.astype(np.int32))
    55         train_err += train_err_batch
    56         train_acc += train_acc_batch

/usr/lib/python2.7/site-packages/theano/compile/function_module.pyc in __call__(self, *args, **kw
887         try:
888             outputs =\
--> 889                 self.fn() if output_subset is None else\
890                 self.fn(output_subset=output_subset)
891         except Exception:

```

KeyboardInterrupt:

6.3.3 Network deleted road, 4 classes, stacked with segs

```

In [9]: with open('data_rem_road.pkl', 'rb') as f:
        X_train, Seg_train, y_train, X_val, Seg_val, y_val, X_test, Seg_test, y_test = pickle.load(

        with open('features_rem_road.pkl', 'rb') as f:
            f1_train, f1_val, f1_test, f2_train, f2_val, f2_test = pickle.load(f)

        f1_train = np.hstack((np.array(f1_train), np.array(f2_train)))
        f1_val = np.hstack((np.array(f1_val), np.array(f2_val)))

        bins = [np.percentile(y_train, 0),
                 np.percentile(y_train, 25),
                 np.percentile(y_train, 50),
                 np.percentile(y_train, 75),
                 np.percentile(y_train, 100) + 1]

        y_train = np.digitize(y_train, bins) - 1
        y_val = np.digitize(y_val, bins) - 1
        y_test = np.digitize(y_test, bins) - 1

        input_layer = InputLayer((None, f1_train[0].shape[0]))
        ll1 = DenseLayer(input_layer, num_units=4096)
        dr1 = DropoutLayer(ll1, p=0.5)
        ll2 = DenseLayer(dr1, num_units=4096)
        dr3 = DropoutLayer(ll2, p=0.5)
        percentile = DenseLayer(dr3, num_units=len(set(y_train)), nonlinearity=softmax)

        target_y = T.vector("target Y integer", dtype='int32')
        input_X = T.matrix('X')

        probs = lasagne.layers.get_output(percentile, input_X, deterministic = True)
        y_predicted = np.argmax(probs, axis=1)

```

```

#all network weights (shared variables)
all_weights = lasagne.layers.get_all_params(percentile)
print(all_weights)

#Mean categorical crossentropy as a loss function - similar to logistic loss but for multiclass
loss = lasagne.objectives.categorical_crossentropy(probs, target_y).mean()
l2_penalty = regularize_layer_params(percentile, l2) * 1e-2
l1_penalty = regularize_layer_params(percentile, l1) * 1e-3
loss += l2_penalty + l1_penalty

#prediction accuracy
accuracy = lasagne.objectives.categorical_accuracy(probs, target_y).mean()

#This function computes gradient AND composes weight updates just like you did earlier
updates_sgd = lasagne.updates.adagrad(loss, all_weights, learning_rate=0.01)

#function that computes loss and updates weights
train_fun = theano.function([input_X, target_y], [loss, accuracy], updates = updates_sgd)

#function that just computes accuracy
accuracy_fun = theano.function([input_X, target_y], accuracy)

num_epochs = 30 #amount of passes through the data

batch_size = 100 #number of samples processed at each function call

for epoch in range(num_epochs):
    # In each epoch, we do a full pass over the training data:
    train_err = 0
    train_acc = 0
    train_batches = 0
    start_time = time.time()
    for batch in iterate_minibatches(f1_train, y_train, batch_size):
        inputs, targets = batch
        train_err_batch, train_acc_batch = train_fun(inputs, targets.astype(np.int32))
        train_err += train_err_batch
        train_acc += train_acc_batch
        train_batches += 1

    # And a full pass over the validation data:
    val_acc = 0
    val_batches = 0
    for batch in iterate_minibatches(f1_val, y_val, batch_size):
        inputs, targets = batch
        val_acc += accuracy_fun(inputs, targets.astype(np.int32))
        val_batches += 1

    # Then we print the results for this epoch:
    print("Epoch {} of {} took {:.3f}s".format(
        epoch + 1, num_epochs, time.time() - start_time))

    print("  training loss (in-iteration):\t\t{:.6f}".format(train_err / train_batches))
    print("  train accuracy:\t\t{:.2f} %".format(

```

```

        train_acc / train_batches * 100))
print(" validation accuracy:\t\t{:.2f} %".format(
    val_acc / val_batches * 100))

```

[W, b, W, b, W, b]

```

-----

MemoryError                                Traceback (most recent call last)

<ipython-input-9-7a80a87eb928> in <module>()
    65     for batch in iterate_minibatches(f1_train, y_train, batch_size):
    66         inputs, targets = batch
--> 67         train_err_batch, train_acc_batch = train_fun(inputs, targets.astype(np.int32))
    68         train_err += train_err_batch
    69         train_acc += train_acc_batch

/home/user/anaconda2/lib/python2.7/site-packages/theano/compile/function_module.py in __call__(self, node, thunk)
    910         node=self.fn.nodes[self.fn.position_of_error],
    911         thunk=thunk,
--> 912         storage_map=getattr(self.fn, 'storage_map', None))
    913     else:
    914         # old-style linkers raise their own exceptions

/home/user/anaconda2/lib/python2.7/site-packages/theano/gof/link.py in raise_with_op(node, thunk)
    312     # extra long error message in that case.
    313     pass
--> 314     reraise(exc_type, exc_value, exc_trace)
    315
    316

/home/user/anaconda2/lib/python2.7/site-packages/theano/compile/function_module.py in __call__(self, node, thunk)
    897     try:
    898         outputs =\
--> 899         self.fn() if output_subset is None else\
    900         self.fn(output_subset=output_subset)
    901     except Exception:

```

```

MemoryError: Error allocating 603979776 bytes of device memory (out of memory).
Apply node that caused the error: GpuDot22(GpuDimShuffle{1,0}.0, GpuElemwise{Composite{(i0 + (i0 * s
Toposort index: 54
Inputs types: [CudaNdarrayType(float32, matrix), CudaNdarrayType(float32, matrix)]
Inputs shapes: [(36864, 100), (100, 4096)]
Inputs strides: [(1, 36864), (4096, 1)]
Inputs values: ['not shown', 'not shown']
Outputs clients: [[GpuElemwise{Sqr}[(0, 0)](GpuDot22.0)]]

```

HINT: Re-running with most Theano optimization disabled could give you a back-trace of when this node was first created.

HINT: Use the Theano flag 'exception.verbosity=high' for a debugprint and storage map footprint of this Apply node.

6.3.4 Network with road, 4 classes, stacked with segs

```
In [8]: with open('data_basic.pkl', 'rb') as f:
        X_train, Seg_train, y_train, X_val, Seg_val, y_val, X_test, Seg_test, y_test = pickle.load(f)

        with open('features_basic.pkl', 'rb') as f:
            f1_train, f1_val, f1_test, f2_train, f2_val, f2_test = pickle.load(f)

        f1_train = np.hstack((np.array(f1_train), np.array(f2_train)))
        f1_val = np.hstack((np.array(f1_val), np.array(f2_val)))

        bins = [np.percentile(y_train, 0),
                 np.percentile(y_train, 25),
                 np.percentile(y_train, 50),
                 np.percentile(y_train, 75),
                 np.percentile(y_train, 100) + 1]

        y_train = np.digitize(y_train, bins) - 1
        y_val = np.digitize(y_val, bins) - 1
        y_test = np.digitize(y_test, bins) - 1

        input_layer = InputLayer((None, f1_train[0].shape[0]))
        l11 = DenseLayer(input_layer, num_units=4096)
        dr1 = DropoutLayer(l11, p=0.5)
        l12 = DenseLayer(dr1, num_units=4096)
        dr3 = DropoutLayer(l12, p=0.5)
        percentile = DenseLayer(dr3, num_units=len(set(y_train)), nonlinearity=softmax)

        target_y = T.vector("target Y integer", dtype='int32')
        input_X = T.matrix('X')

        probs = lasagne.layers.get_output(percentile, input_X, deterministic = True)
        y_predicted = np.argmax(probs, axis=1)

        #all network weights (shared variables)
        all_weights = lasagne.layers.get_all_params(percentile)
        print(all_weights)

        #Mean categorical crossentropy as a loss function - similar to logistic loss but for multiclass
        loss = lasagne.objectives.categorical_crossentropy(probs, target_y).mean()
        l2_penalty = regularize_layer_params(percentile, l2) * 1e-1
        l1_penalty = regularize_layer_params(percentile, l1) * 1e-2
        loss += l2_penalty + l1_penalty

        #prediction accuracy
        accuracy = lasagne.objectives.categorical_accuracy(probs, target_y).mean()

        #This function computes gradient AND composes weight updates just like you did earlier
        updates_sgd = lasagne.updates.adagrad(loss, all_weights, learning_rate=0.01)

        #function that computes loss and updates weights
        train_fun = theano.function([input_X, target_y], [loss, accuracy], updates = updates_sgd)

        #function that just computes accuracy
```

```

accuracy_fun = theano.function([input_X, target_y], accuracy)

num_epochs = 30 #amount of passes through the data

batch_size = 100 #number of samples processed at each function call

for epoch in range(num_epochs):
    # In each epoch, we do a full pass over the training data:
    train_err = 0
    train_acc = 0
    train_batches = 0
    start_time = time.time()
    for batch in iterate_minibatches(f1_train, y_train, batch_size):
        inputs, targets = batch
        train_err_batch, train_acc_batch = train_fun(inputs, targets.astype(np.int32))
        train_err += train_err_batch
        train_acc += train_acc_batch
        train_batches += 1

    # And a full pass over the validation data:
    val_acc = 0
    val_batches = 0
    for batch in iterate_minibatches(f1_val, y_val, batch_size):
        inputs, targets = batch
        val_acc += accuracy_fun(inputs, targets.astype(np.int32))
        val_batches += 1

    # Then we print the results for this epoch:
    print("Epoch {} of {} took {:.3f}s".format(
        epoch + 1, num_epochs, time.time() - start_time))

    print("  training loss (in-iteration):\t\t{:.6f}".format(train_err / train_batches))
    print("  train accuracy:\t\t{:.2f} %".format(
        train_acc / train_batches * 100))
    print("  validation accuracy:\t\t{:.2f} %".format(
        val_acc / val_batches * 100))

[W, b, W, b, W, b]
Epoch 1 of 30 took 4.315s
  training loss (in-iteration):          4726.092065
  train accuracy:          24.38 %
  validation accuracy:          33.00 %
Epoch 2 of 30 took 4.367s
  training loss (in-iteration):          11.885494
  train accuracy:          30.03 %
  validation accuracy:          26.00 %
Epoch 3 of 30 took 4.157s
  training loss (in-iteration):          5.118199
  train accuracy:          30.03 %
  validation accuracy:          25.20 %
Epoch 4 of 30 took 4.156s
  training loss (in-iteration):          4.719048
  train accuracy:          33.24 %

```

validation accuracy:	25.00 %	
Epoch 5 of 30 took 4.154s		
training loss (in-iteration):		4.602650
train accuracy:	33.83 %	
validation accuracy:	25.40 %	
Epoch 6 of 30 took 4.158s		
training loss (in-iteration):		4.511264
train accuracy:	34.45 %	
validation accuracy:	29.20 %	
Epoch 7 of 30 took 4.156s		
training loss (in-iteration):		4.433514
train accuracy:	35.31 %	
validation accuracy:	31.80 %	
Epoch 8 of 30 took 4.153s		
training loss (in-iteration):		4.364174
train accuracy:	35.90 %	
validation accuracy:	33.00 %	
Epoch 9 of 30 took 4.321s		
training loss (in-iteration):		4.301553
train accuracy:	36.66 %	
validation accuracy:	34.60 %	
Epoch 10 of 30 took 4.163s		
training loss (in-iteration):		4.242831
train accuracy:	38.03 %	
validation accuracy:	36.00 %	
Epoch 11 of 30 took 4.158s		
training loss (in-iteration):		4.186211
train accuracy:	40.17 %	
validation accuracy:	37.40 %	
Epoch 12 of 30 took 4.189s		
training loss (in-iteration):		4.131390
train accuracy:	42.07 %	
validation accuracy:	37.60 %	
Epoch 13 of 30 took 4.163s		
training loss (in-iteration):		4.075547
train accuracy:	43.79 %	
validation accuracy:	39.00 %	
Epoch 14 of 30 took 4.183s		
training loss (in-iteration):		4.020848
train accuracy:	45.31 %	
validation accuracy:	40.00 %	
Epoch 15 of 30 took 4.165s		
training loss (in-iteration):		3.965067
train accuracy:	47.62 %	
validation accuracy:	42.20 %	
Epoch 16 of 30 took 4.164s		
training loss (in-iteration):		3.908799
train accuracy:	50.52 %	
validation accuracy:	42.60 %	
Epoch 17 of 30 took 4.170s		
training loss (in-iteration):		3.857314
train accuracy:	53.24 %	
validation accuracy:	43.60 %	
Epoch 18 of 30 took 4.167s		

training loss (in-iteration):		3.806450
train accuracy:	55.59 %	
validation accuracy:	43.20 %	
Epoch 19 of 30 took 4.179s		
training loss (in-iteration):		3.757564
train accuracy:	57.03 %	
validation accuracy:	43.20 %	
Epoch 20 of 30 took 4.165s		
training loss (in-iteration):		3.713492
train accuracy:	58.41 %	
validation accuracy:	42.40 %	
Epoch 21 of 30 took 4.181s		
training loss (in-iteration):		3.673273
train accuracy:	59.66 %	
validation accuracy:	42.20 %	
Epoch 22 of 30 took 4.169s		
training loss (in-iteration):		3.635788
train accuracy:	60.59 %	
validation accuracy:	43.00 %	
Epoch 23 of 30 took 4.188s		
training loss (in-iteration):		3.601693
train accuracy:	61.66 %	
validation accuracy:	40.60 %	
Epoch 24 of 30 took 4.169s		
training loss (in-iteration):		3.569888
train accuracy:	63.17 %	
validation accuracy:	41.60 %	
Epoch 25 of 30 took 4.192s		
training loss (in-iteration):		3.539712
train accuracy:	63.79 %	
validation accuracy:	41.00 %	
Epoch 26 of 30 took 4.172s		
training loss (in-iteration):		3.510056
train accuracy:	64.86 %	
validation accuracy:	41.40 %	
Epoch 27 of 30 took 4.164s		
training loss (in-iteration):		3.482480
train accuracy:	66.41 %	
validation accuracy:	40.20 %	
Epoch 28 of 30 took 4.187s		
training loss (in-iteration):		3.455721
train accuracy:	67.69 %	
validation accuracy:	40.80 %	
Epoch 29 of 30 took 4.164s		
training loss (in-iteration):		3.428972
train accuracy:	68.83 %	
validation accuracy:	41.20 %	
Epoch 30 of 30 took 4.173s		
training loss (in-iteration):		3.419649
train accuracy:	68.41 %	
validation accuracy:	41.20 %	

7 Results

The best score was achieved with stacking technique.

```
In [23]: from stacking import Stacking
```

```
basic_wildfowl = Stacking(base_estimators=[
    (LogisticRegression(C=0.01, penalty='l1').fit,
     lambda clf, X: clf.predict(X)),
    (lambda X, y: LinearSVC(C=1, penalty='l2', loss='squared_hinge',
                           dual=True, multi_class='crammer_singer').fit(normalizer.transform(X)),
     lambda clf, X: clf.predict(normalizer.transform(X))),
    (lambda X, y: LogisticRegression(C=8, penalty='l1').fit(normalizer.transform(X), y),
     lambda clf, X: clf.predict(normalizer.transform(X))),
    (lambda X, y: SVC(C=1, kernel='rbf', gamma=1).fit(normalizer.transform(X), y),
     lambda clf, X: clf.predict(normalizer.transform(X))),
    (lambda X, y: SVC(C=1, kernel='poly', gamma=1, degree=2).fit(normalizer.transform(X), y),
     lambda clf, X: clf.predict(normalizer.transform(X)))],
    meta_fitter=SVC(C=1, kernel='rbf', gamma=0.2).fit,
    n_folds=5, extend_meta=False)

y_predicted = basic_wildfowl.fit(np.array(f1_train), y_train).predict(np.array(f1_val))
```

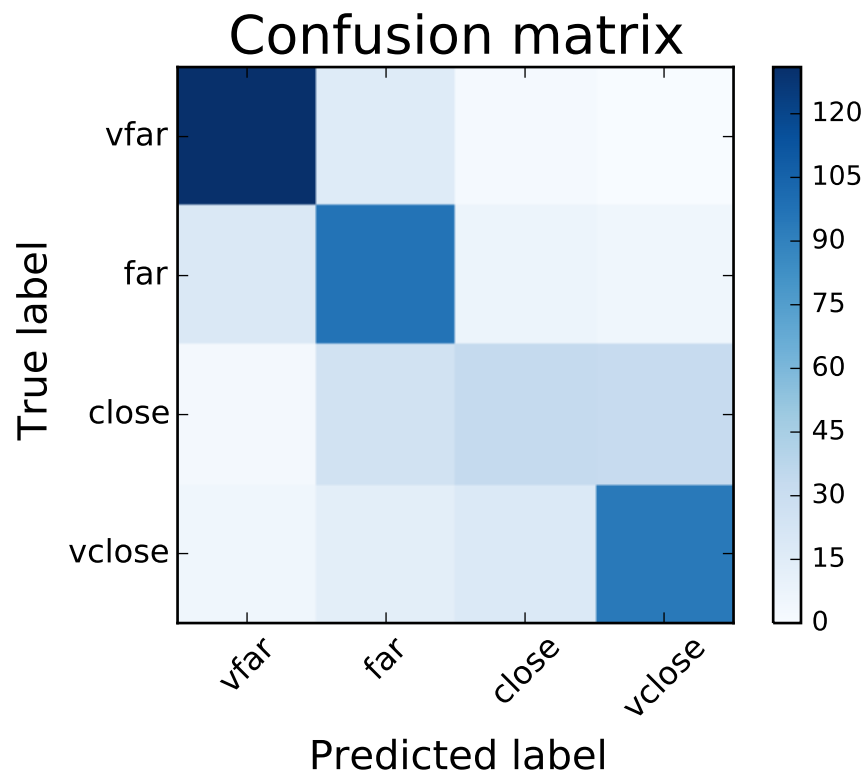
```
/usr/lib64/python2.7/site-packages/sklearn/cross_validation.py:69: DeprecationWarning: The indices parameter
stacklevel=1)
```

```
In [24]: from sklearn.metrics import confusion_matrix
```

```
def plot_confusion_matrix(cm, title='Confusion matrix', cmap=plt.cm.Blues):
    plt.figure(figsize=(5, 4))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title, size=20)
    plt.colorbar()
    labels = ['vfar', 'far', 'close', 'vclose']
    tick_marks = np.arange(len(labels))
    plt.xticks(tick_marks, labels, rotation=45, size=12)
    plt.yticks(tick_marks, labels, size=12)
    plt.tight_layout()
    plt.ylabel('True label', size=15)
    plt.xlabel('Predicted label', size=15)
    plt.savefig('cm.pdf')
    plt.show()
```

```
plot_confusion_matrix(confusion_matrix(y_val, y_predicted))
```

```
/usr/lib64/python2.7/site-packages/matplotlib/collections.py:590: FutureWarning: elementwise comparison
if self._edgecolors == str('face'):
```



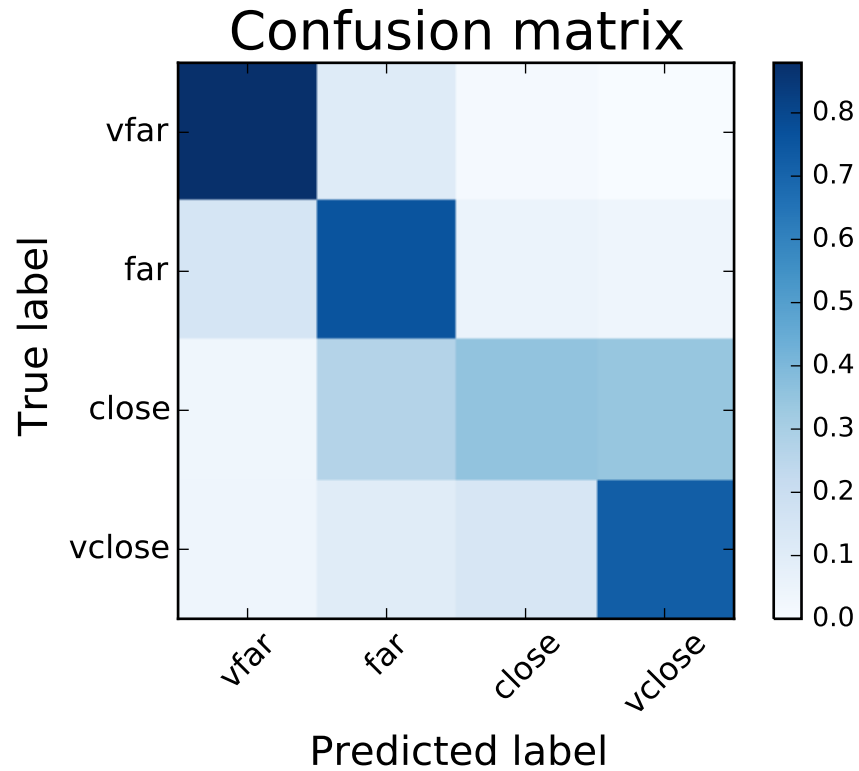
```
In [25]: A = confusion_matrix(y_val, y_predicted)
         np.set_printoptions(precision=2)
         print('Confusion matrix, without normalization')
         cm = A.astype(float) / A.sum(1, keepdims=True)
         print(cm)
         plt.figure()
         plot_confusion_matrix(cm)
```

Confusion matrix, without normalization

```
[[ 0.88  0.11  0.01  0. ]
 [ 0.15  0.76  0.05  0.04]
 [ 0.03  0.27  0.35  0.34]
 [ 0.04  0.1  0.14  0.72]]
```

<matplotlib.figure.Figure at 0x18b410d0>

Segmentation	+		-	
Road	+	-	+	-
Neural Network	66.8%	NA	67.7%	66.7%
Logistic Regression	67.6%	66.6%	NA	68.0%
Normalization + LogReg	67.7%	NA	NA	67.0%
Linear SVC	67.6%	NA	NA	67.0%
PCA + SGB	58.8%	NA	59.7%	NA
Stacking of all above	NA %	NA	NA%	70.6%



8 Conclusion

While working on the project we:

- Accomplished Data Processing included segmentation (SegNet) and disparity map building (SGBM), noise reduction
- Developed and Experimented with Classifier based on features extracted from VGG19

From the confusion matrix it can be seen, that there are quite good results for the most interesting for us area (very close). Results:

9 Individual contribution

- Data preparation: Anastasia Makarova and others
- ML, Classifiers: Mikhail Karasikov
- External segmentation with SegNet and ANN training: Mikhail Usvyatsov
- All the other labor: All together