Projet INF442
Object Tracking using Convolutional Neural Networks

João Loula

joao.campos-loula@polytechnique.edu

X2014
Session 2016

1 Introduction

Neural Networks

There has recently been a lot of hype around Deep Neural Networks, stemming specially from their outstanding results in computer vision tasks (most notably since the seminal paper (Krizhevsky et al., 2012)), but also in areas like machine translation and speech recognition. Although their unreasonable effectiveness continues to baffle computer scientists and mathematicians alike, the principles guiding their behavior are fairly uncomplicated: at its simplest, a neural network is an algorithm comprised of an *Input* layer, an *Output* layer, and any number of so-called *Hidden* layers in between (figure 1). The latter map an input $x \in \mathbb{R}^n$ to an output $y \in \mathbb{R}^m$ by the following transformation:

$$y = f\left(Wx + b\right)$$

where $W \in M_{m,n}$ (the *weight*) is a linear transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$, $b \in \mathbb{R}^m$ (the bias) is a vector and f (the activation function) is some continuous function (usually chosen to be the sigmoid, $tanh$ or x^+).

The parameters to be learned in the model are the weights and biases of the network: this can for example be done through gradient descent, more specifically through an implementation called *backpropagation*. The idea is that, in order to compute the optimal change in parameters to minimize some cost function $J(y^*, y)$ dependent on the output y^* of the network and the ground-truth y, though we do not know its dependence on each parameter explicitly, we do know how y^* depends on the parameters of the final layer (of which it is the output), and we know how the input to the final layer depends on the parameters of the layer before etc. Using the chain-rule, we can then propagate this gradient back through each layer, and thus calculate the optimal change in parameters [1].

[1] In practice, the datasets used are too large for their gradient to be computed all at once: we choose then to do iterations of backpropagation on batches randomly sampled from the dataset, in a method called *stochastic gradient descent*.

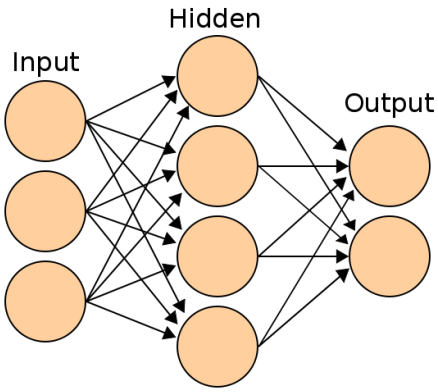

Figure 1: Schema of a Neural Network (source: Wikipedia).

CNNs

Convolutional Neural Networks (CNNs) are a special flavor of neural networks in which the basic operation is matrix convolution. The interest of this setup is twofold: by having a layer convolute some kernel with the input, we reduce the number of parameters that need to be learned (we can think of the convolution as applying various copies of the same transformation to the input) and we also give the network translation invariance, which is great for capturing 2D structure, for example in computer vision tasks [2].

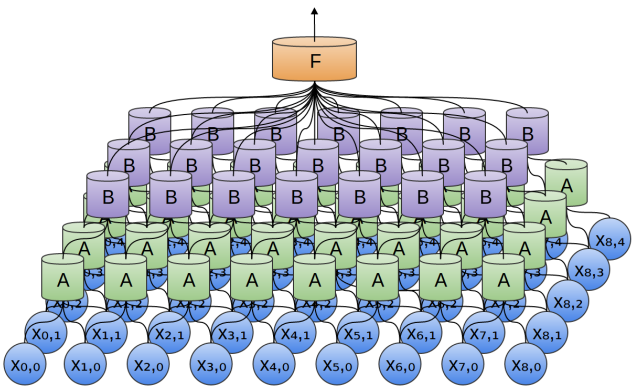

Figure 2: Visualization of a two-dimensional CNN : the convolution operation can be thought of as the action of multiple copies of the same neuron on the input (source: Christopher Olah's blog).

Tracking

The goal of this project is to use an architecture called Siamese CNNs to solve an object tracking problem, that is, to map the location of a given object through

[2]While 2D are preponderant, it should be noted that higher dimension networks have also found uses in video and medical image analysis, and 1D networks are implemented for example for classification by music streaming services like spotify.

time in video data, a central problem in areas like autonomous vehicle control and motion-capture videogames.

Siamese CNNs (Chopra et al., 2005) are a model consisting of two identical CNNs that share all their weights. We can think of them as embedding two inputs into some highly structured space, where this output can then be used by some other function. Notable examples include using Siamese CNNs to determine, given two photos, whether they represent the same person (Sun et al., 2014) or, given two images taking consecutively by a moving vehicle, determine the translational and rotational movements that the vehicle has performed (Agrawal et al., 2015).

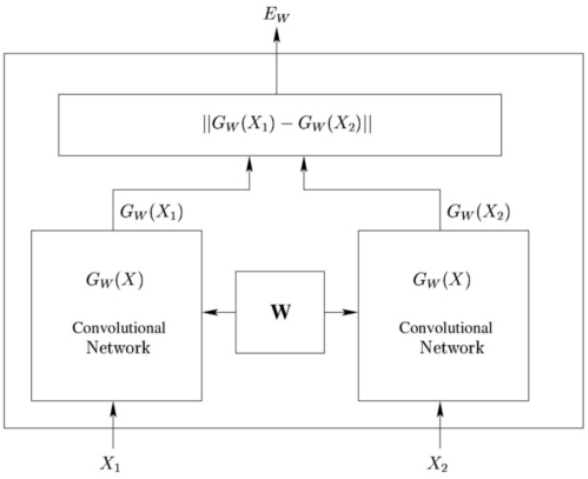

Figure 3: Visualization of the Siamese CNNs architecture: the two CNNs are identical and share all their weights. In this scheme, their output is directed to an energy function that calculates the norm of the difference (source: (Chopra et al., 2005)).

The idea of the implementation is to train the Siamese CNNs model on evenly spaced pairs of frames in a video of an object moving, and to feed their output to another network that will try to learn the object's movement between the two frames.

Figure 4: Example of two video frames to serve as input to the Siamese CNNs model: the bounding box represent the ground-truth of the object movement in the dataset (source: (Mei and Porikli, 2008)).

2 A Sip of Caffe

Caffe (Jia et al., 2014) is a deep learning framework written in and interfaced with C++, created by the Berkeley Vision and Learning Center. At its core, it is based on two main objects :

- *Nets* represent the architecture of the deep neural network : they are comprised of *layers* of different types (convolutional, fully-connected, dropout etc.) ;
- *Blobs* are simply C++ arrays : the data structures being passed along the nets.

Blobs are manipulated throughout the net in *forward* and *backward* passes : forward passes denote the process in which the neural network takes some data as input and outputs a prediction, while backward passes refer to backpropagation : the comparison of this prediction with the label and the computation of the gradients of the loss function with respect to the parameters throughout the network in a backwards fashion.

Models

One of the great strengths of Caffe is the fact that its models are stored in plaintext Google Protocol Buffer (Google) schemas : it is highly serializable and human-readable, and interfaces well with many programming languages (such as C++ and Python). Let's take a look at how to declare a convolution layer in protobuf:

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
        name: "conv1_w"
        lr_mult: 1
```

```
  }
  param {
        name: "conv1_b"
        lr_mult: 2
  }
  convolution_param {
    num_output: 256
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

"Name" and "Type" are very straightforward entries : they define a name and a type for that layer. "Bottom" and "Top" define respectively the input and output of the layer. The "param" section defines rules for the parameters of the layers (weights and biases) : the "name" section will be of utmost importance in this project, since naming the parameters will allow us to share them through networks and thus realize the Siamese CNNs architecture, and "lr_mult" defines the multipliers of the learning rates for the parameters (making the biases change twice as fast as the weights tends to work well in practice).

Parallelisation

MPI-Caffe (Lee et al., 2015) is a framework built by a group at the University of Indiana to interface MPI with Caffe. By default it parallelizes all layers of the network through all nodes in the cluster : nodes can be included or excluded from computation in specific layers. Communication processes like MPIBroadcast and MPIGather are written as layers in the .protobuf file, and the framework automatically computes the equivalent expression for the gradients in the backward pass.

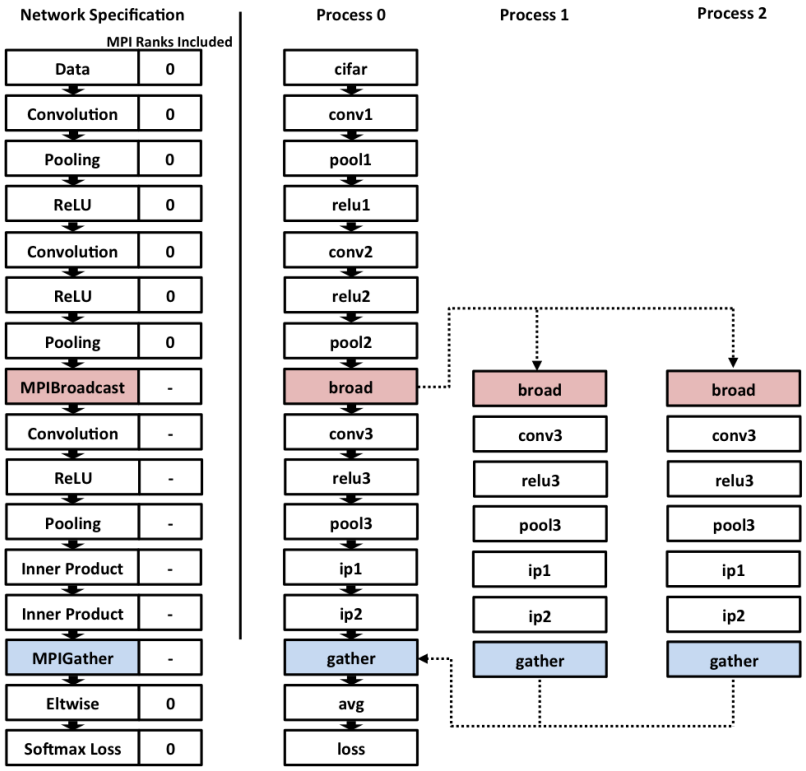

Figure 5: Example of a CNN architecture parallelised using MPI-Caffe. The Network is specified on the left, and for each layer there is a "0" when only the root is included in that layer's computation and a "-" when all nodes are included in it. The MPIBroadcast and MPIGather begin and end respectively the parallelised section of the code (source: (Lee et al., 2015)).

One of the great advantages of the model is that possibility of parallelisation is twofold:

1. *Across Siamese Networks* (medium grain): the calculations performed by each of the two Siamese CNNs can be run independently, with their results being sent back to feed the function on top;

2. *Across Image Pairs* (coarse grain): to increase the number of image pairs in each batch in training, and the speed with which they are processed, we can separate them in mini-batches that are processed across different machines in a cluster.

3 MNIST

The Dataset

MNIST (LeCun et al., 1999) is a dataset consisting of 70,000 28x28 grayscale images (split in a train and a test set in a 6:1 proportion) representing handwritten digits, with labels from 0 to 9 that stand for the digit represented by

each image. The dataset is stored in the not-so-intuitive IDX file format, but we'll be using a CSV version available online in this project.

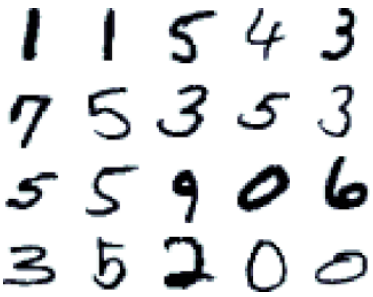

Figure 6: Example of images from the MNIST dataset (source: Rodrigo Benenson's blog).

Preprocessing

For the tracking task, preprocessing was done by transforming images in the dataset by a combination of rotations and translations. Rotations were restrained to 3° intervals in $[-30°, 30°]$, and translations were chosen as integers in $[-3, 3]$.

The task to be learned was posed as classification over the set of possible rotations and translations, with the loss function being the sum of the losses for rotation, x-axis translation and y-axis translation.

The Network

Using the nomenclature BCNN (for Base Convolutional Neural Network) for the architecture of the Siamese networks and TCNN (for Top Convolutional Neural Network) for the network that takes input from the Siamese CNNs and outputs the final prediction, the architecture used was the following (see figure 14 on the annex) :

- BCNN :
  - A convolution layer, with 3x3 kernel and 96 filters, followed by ReLU nonlinearity;
  - A 2x2 max-pooling layer;
  - A convolution layer, with 3x3 kernel and 256 filters, followed by ReLU;
  - A 2x2 max-pooling layer;
- TCNN :
  - A fully-connected layer, with 500 filters, followed by ReLU nonlinearity;
  - A dropout layer with 0.5 dropout;
  - Three separate fully-connected layers, with 41, 13 and 13 outputs respectively (matching number of rotation, x translation and y translation classes);

– A softmax layer with logistic loss (with equal weights for each of the three predictions).

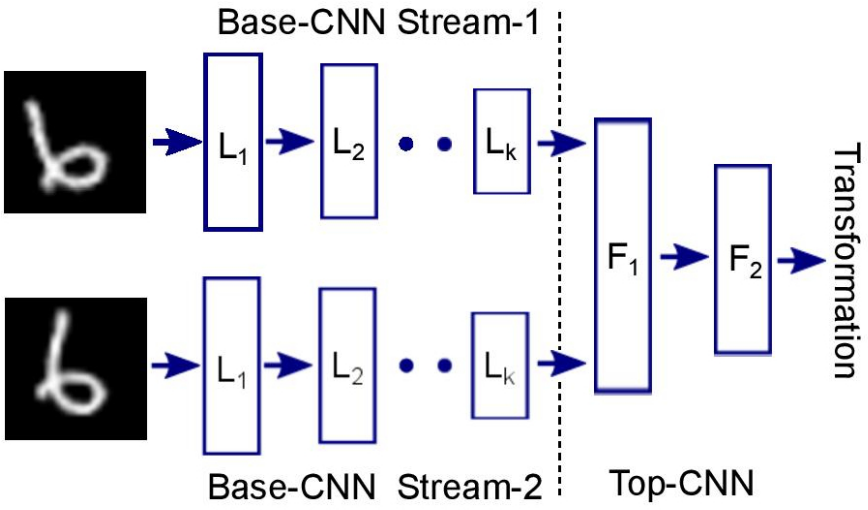

Figure 7: Scheme of a forward pass in the Siamese network: each image in the pair moves through the layers L1, ... Lk in one of the BCNNs, and their output is processed by the TCNN to make the prediction (source: (Agrawal et al., 2015)).

Results

The network was trained using batches of 64 image pairs, with a base learning rate of 10^{-7} and inverse decay with $\gamma = 0.1$ and power = 0.75. The network seemed to converge after about 1000 iterations, to an accuracy of about 3%for the rotation prediction and 14% for the x and y translation predictions (about 1.25 times better than random guessing for the rotation and 2 times better for the translations).

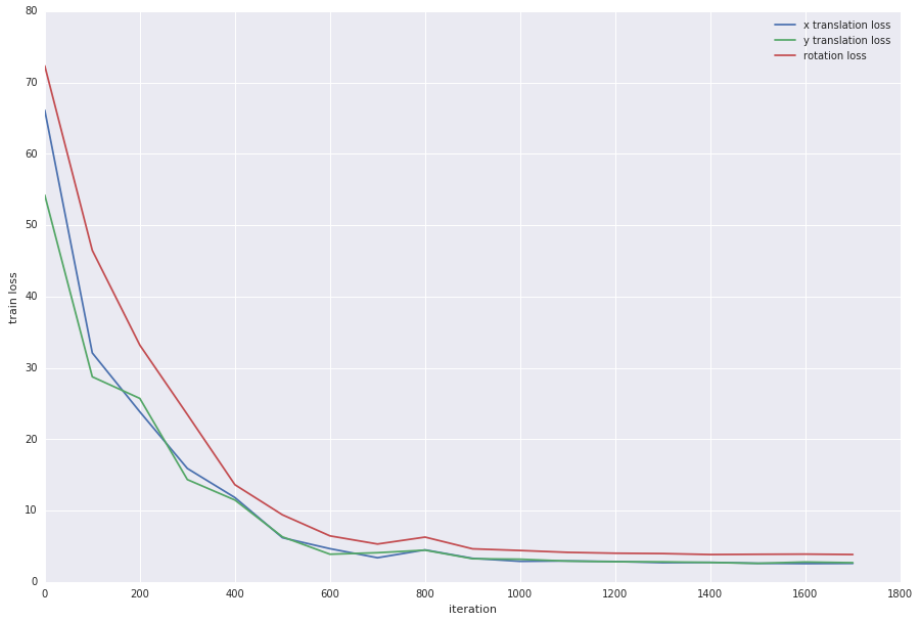

Figure 8: Value of the loss function throughout training iterations in the model.

Coarse-Grain Parallelization

The simplest way to parallelize the program is to run multiple training batches on different nodes, as in the scheme in figure 9.

In this case, we're gaining a speedup in the Gustafson sense, that is, as we raise the number of processors, we also raise the size of the data we can compute in a given time. The speedup expression is then given by:

$$\text{speedup}_{\text{Gustafson}}(P) = \alpha_{seq} + P(1 - \alpha_{seq})$$

where P is the number of processors and α_{seq} is the proportion of the code that's not being parallelized. Seeing as in this scheme the whole network is being parallelized, we have:

$$\alpha_{seq} \approx 0 \Rightarrow \text{speedup}_{\text{Gustafson}}(P) \approx P$$

Let's see how this fares in practice. In figure 10, we find a comparison of running times for the forward and backward passes in the network for one, two and four cores, the four core option using hyperthreading. What we find is that the two core case follows Gustafson's law closely, with a speedup coefficient of 1.93. In the four core case, however, performance is no better than with two cores, which probably means that hyperthreading is making no difference for this task.

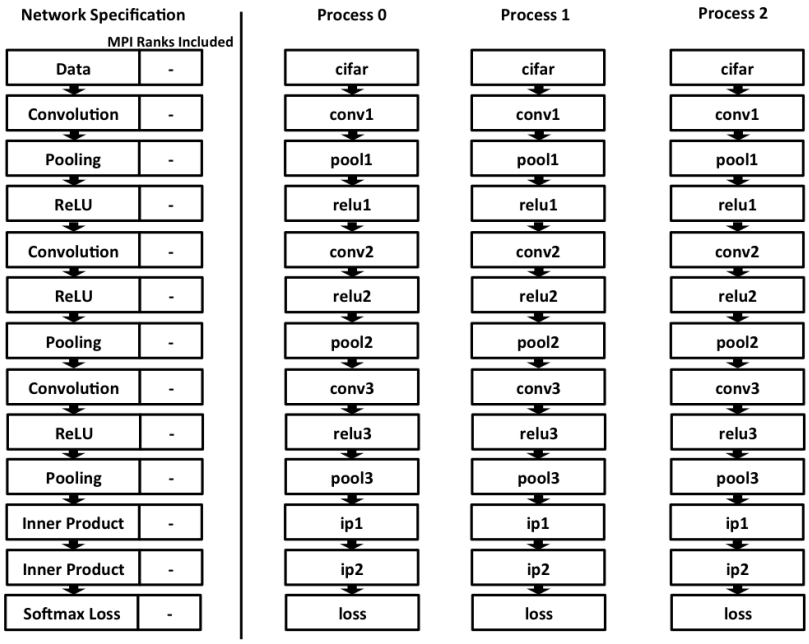

Figure 9: Example of a CNN architecture using fully parallelised using MPI-Caffe (source: Lee et al. (2015)).

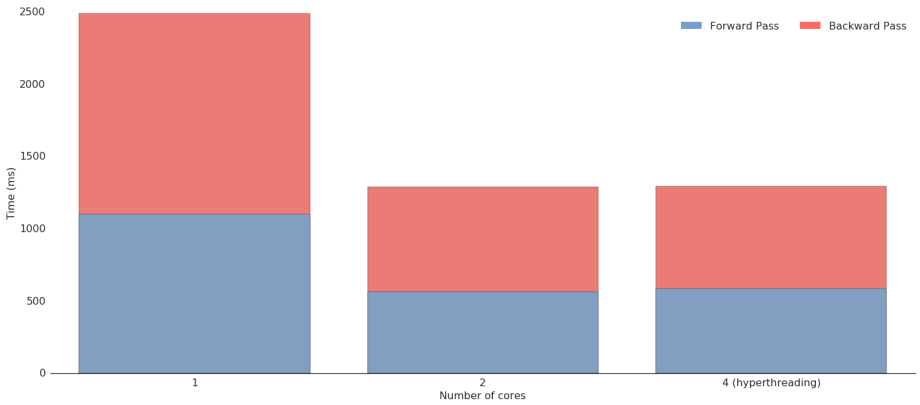

Figure 10: Comparison between forward and backward pass times when running the network with 1, 2 or 4 cores with hyperthreading.

Medium-Grain Parallelization

The interest of the Siamese CNNs architecture, however, is the possibility of parallelization on a lower level : we can distribute the two BCNN streams to two different nodes in the cluster, and then gather their results to perform the computations on the TCNN, in a way similar to that shown in figure 5. Results are shown in figure 11: we can see that performance is almost as good as in the completely parallelized scheme, which confirms our knowledge that

the convolutional layers are by far the most computationally-intensive ones, so that the BCNN accounts for most of the computations in the network. We can also see that the difference between these two parallelization schemes lies almost entirely in the backward pass: we can hypothesize that this is due to increased difficulty in computing the gradient through the gather and broadcast layers in the Medium-Grain scheme.

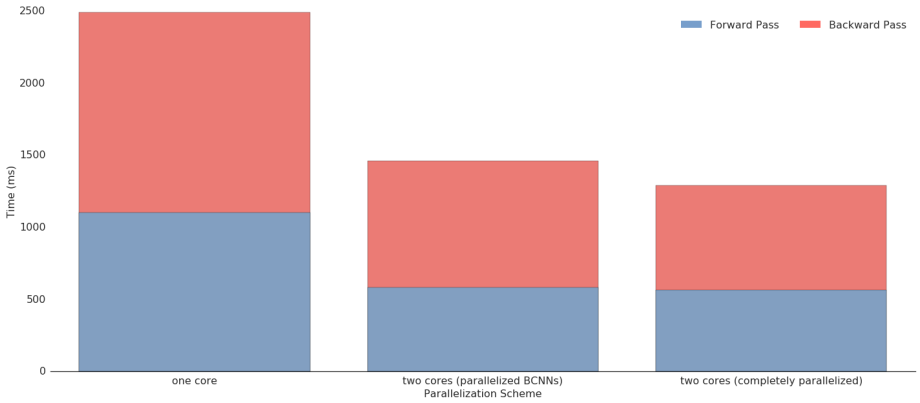

Figure 11: Comparison between forward and backward pass times when running the network with no parallelization, with only the BCNN parallelized or with the whole code parallelized.

4 KITTI

The Dataset

The KITTI odometry dataset (Geiger et al., 2012) (product of a collaboration between the Max Planck Institute at Tübingen,) is comprised of video frames and odometry information collected from 11 different trips made by an autonomous car.

Figure 12: Example of image pair from the KITTI dataset in grayscale.

The Network

Using the same nomenclature as for the MNIST dataset :

- BCNN :
  - A convolution layer, with 3x3 kernel and 96 filters, followed by ReLU nonlinearity and 2x2 max-pooling;
  - A convolution layer, with 3x3 kernel and 256 filters, followed by ReLU and 2x2 max-pooling;
  - A convolution layer, with 3x3 kernel and 384 filters, followed by ReLU;
  - A convolution layer, with 3x3 kernel and 384 filters, followed by ReLU;
  - A convolution layer, with 3x3 kernel and 256 filters, followed by ReLU and 2x2 max-pooling;
- TCNN :
  - A convolution layer, with 3x3 kernel and 256 filters, followed by ReLU and 2x2 max-pooling;
  - A convolution layer, with 3x3 kernel and 128 filters, followed by ReLU and 2x2 max-pooling;
  - A fully-connected layer, with 500 filters, followed by ReLU nonlinearity;
  - A dropout layer with 0.5 dropout;
  - A softmax layer with logistic loss (with equal weights for each of the three predictions).

Results

Coming soon!

5 Conclusion

With long training times and the increasing number of real-time applications, parallelization has become a necessity for neural networks. While a lot of recent efforts seem centered around fine-grain approaches (e.g. distributed stochastic gradient descent), we tried to show here the merit of medium-grain and coarse-grain schemes for Siamese CNNs, which seem to be a match made in heaven, and present some interesting techniques and results. This sort of scheme could be of great use for example if exploited by tracking systems that must learn to guide themselves in an environment on-the-fly, like rescue drones, such that the ability to process inputs faster would represent a significant upgrade in their effectiveness.

References

P. Agrawal, J. Carreira, and J. Malik. Learning to see by moving. *arXiv preprint*, 2015.

S. Chopra, R. Hadsell, and Y. LeCun. Learning a similarity metric discriminatively, with application to face verification. In *Proc. CVPR*, 2005.

Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

Google. Protocol buffers. `http://code.google.com/apis/protocolbuffers/`.

Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *ACM Multimedia Open Source*, 2014.

A. Krizhevsky, Ilya Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Proc. NIPS*, 2012.

Y. LeCun, C. Cortes, and C. J.C. Burges. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist*, 1999.

Stefan Lee, Senthil Purushwalkam, Michael Cogswell, David J. Crandall, and Dhruv Batra. Why M heads are better than one: Training a diverse ensemble of deep networks. *arXiv*, 2015. URL `http://arxiv.org/abs/1511.06314`.

X. Mei and F. Porikli. Joint tracking and video registration by factorial hidden markov models. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2008.

Y. Sun, X. Wang, and X. Tang. Deep learning face representation by joint identification-verification. *Proc. NIPS*, 2014.

Annex

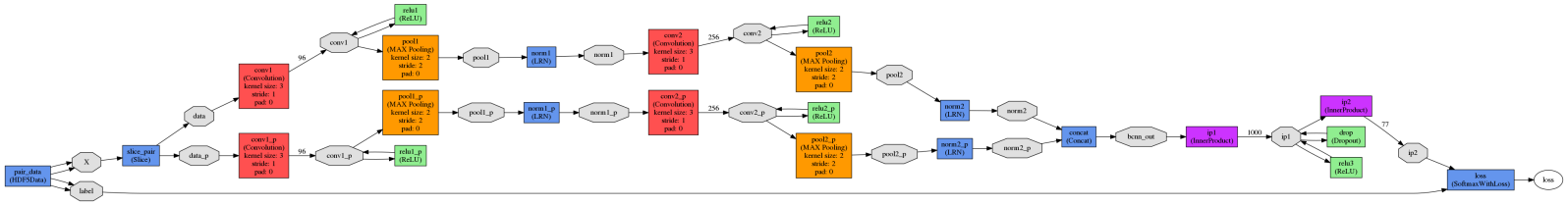

Figure 13: Representation of the siamese model used for the mnist dataset.

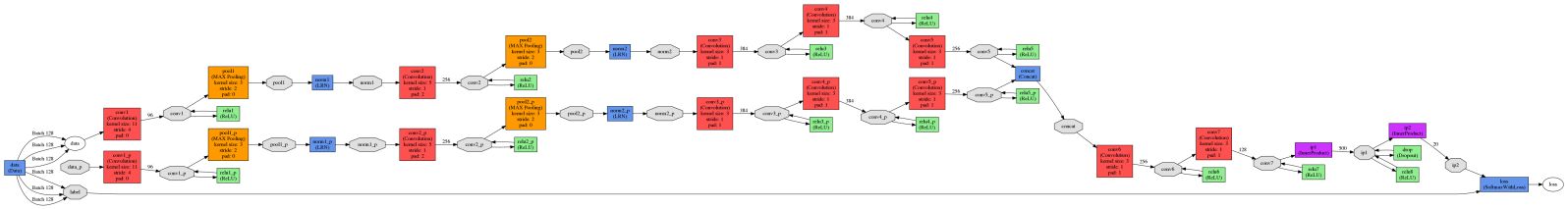

Figure 14: Representation of the siamese model used for the kitti dataset.