Final Project ML

Sidorin Aleksandr

The link to Colab https://colab.research.google.com/drive/1okXBm86xZDzCC8Y8HcQVa1AXpwSVCss8

Paper http://jmlr.org/papers/volume17/15-239/15-239.pdf

The task is to build a model classifier that gives good result on MNIST, but train on SVHN and use domain adaptation.

1. My model:

In this task I am going to implement approach **(from a paper)**, which is directly inspired by the theory on domain adaptation suggesting that, for effective domain transfer to be achieved, predictions must be made based on features that can not discriminate between the training (source) and test (target) domains.

Reading this paper I focus on learning features that combine discriminativeness and domain invariance. This is achieved by jointly optimizing the underlying features and two discriminative classifiers operating on these features:

- the label predictor that predicts class labels and is used both during training and at testing time;

- the domain classifier that discriminates between the source and the target domains during training.

While the parameters of the classifiers are optimized in order to minimize their error on the training set, the parameters of the underlying deep feature mapping are optimized in order to minimize the loss of the label classifier and to maximize the loss of the domain classifier. The latter update encourages domain invariant features to emerge in the course of the optimization.

Also in my model I have one non-standard component - trivial gradient reversal layer that leaves the input unchanged during forward propagation

and reverses the gradient by multiplying it by a negative scalar during the backpropagation.

In my model I used the next definition: I compute the empirical H-divergence between two samples by running algorithm on the problem of discriminating between source and target examples. I constructed a new data set where the examples of of the source sample are labeled *0* and examples of the target sample are labelled *1*.

$$U = \{(\mathbf{x}_i, 0)\}_{i=1}^{n} \cup \{(\mathbf{x}_i, 1)\}_{i=n+1}^{N}$$

The *H*-divergence is then approximated by

$$\hat{d}_{\mathcal{A}} = 2\,(1 - 2\epsilon)$$

Training the neural network leads to the optimization problem on the source domain:

$$\min_{\mathbf{W},\mathbf{b},\mathbf{V},\mathbf{c}} \left[\frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_y^i(\mathbf{W}, \mathbf{b}, \mathbf{V}, \mathbf{c}) + \lambda \cdot R(\mathbf{W}, \mathbf{b}) \right]$$

where $L^i_y(W, b, V, c) = L_y G_y(G_f(x_i; W, b); V, c)$, y_i is a shorthand notation for the prediction.

Loss on the *i-th* example.

For the examples from the target domains, I do not know the labels at training time, and I would like to predict such labels at test time. This enables me to add a domain adaptation term to the objective of Equation, giving the following regularizer:

$$R(\mathbf{W}, \mathbf{b}) = \max_{\mathbf{u},z} \left[-\frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_d^i(\mathbf{W}, \mathbf{b}, \mathbf{u}, z) - \frac{1}{n'} \sum_{i=n+1}^{N} \mathcal{L}_d^i(\mathbf{W}, \mathbf{b}, \mathbf{u}, z) \right]$$

where $L^i_d(W, b, u, z) = L_d G_d(G_f(x_i; W, b); u, z), d_i)$

The hyper parameter λ is then used to tune the trade-off between two quantities during learning process.

For learning algorithm I have the complete optimization objective:

$$E(\mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c}, \mathbf{u}, z) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_y^i(\mathbf{W}, \mathbf{b}, \mathbf{V}, \mathbf{c}) - \lambda\left(\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_d^i(\mathbf{W}, \mathbf{b}, \mathbf{u}, z) + \frac{1}{n'}\sum_{i=n+1}^{N}\mathcal{L}_d^i(\mathbf{W}, \mathbf{b}, \mathbf{u}, z)\right)$$

where

we are seeking the parameters W, V, b, c, u, z that deliver a saddle point given by

$$\begin{aligned}(\hat{\mathbf{W}}, \hat{\mathbf{V}}, \hat{\mathbf{b}}, \hat{\mathbf{c}}) &= \underset{\mathbf{W},\mathbf{V},\mathbf{b},\mathbf{c}}{\operatorname{argmin}} E(\mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c}, \hat{\mathbf{u}}, \hat{z}) \\ (\hat{\mathbf{u}}, \hat{z}) &= \underset{\mathbf{u},z}{\operatorname{argmax}} E(\hat{\mathbf{W}}, \hat{\mathbf{V}}, \hat{\mathbf{b}}, \hat{\mathbf{c}}, \mathbf{u}, z)\end{aligned}$$

There is an algorithm Stochastic training update in Application

The difference from 1st submission:

In 1st submission I had only one simple CNN model with *3* layers, and I did not use target images (MNIST). Now I have CNN model with Feature Extractor, Domain Classifier and Label Predictor. Now I have hyper parameters *p* and *lambda*.

How did I tune them?

I tuned them using formulas from paper.

p is changing from *0* to *1* during training process.

p = (batch_idx + start_steps) / total_steps,

where *start_steps = current epoch number * length of train array, total_steps = number of all epoch * length of train array*

The domain adaptation parameter λ is initiated at *0* and is gradually changed to *1* using the schedule:

$$\lambda_p = \frac{2}{1 + \exp(-\gamma \cdot p)} - 1$$

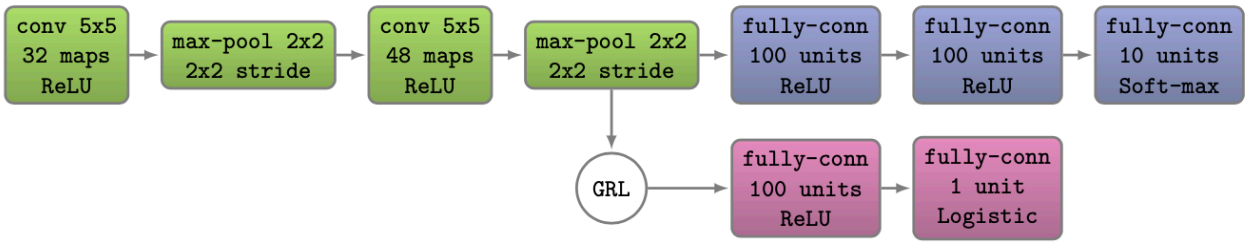
where *γ* is set to *10*.

The learning rate is adjusted during the stochastic gradient descent using the formula:

$$\mu_p = \frac{\mu_0}{(1 + \alpha \cdot p)^\beta}$$

where p is the training progress linearly changing from 0 to 1, $\mu_0 = 0.01$, $\alpha = 10$ and $\beta = 0.75$.

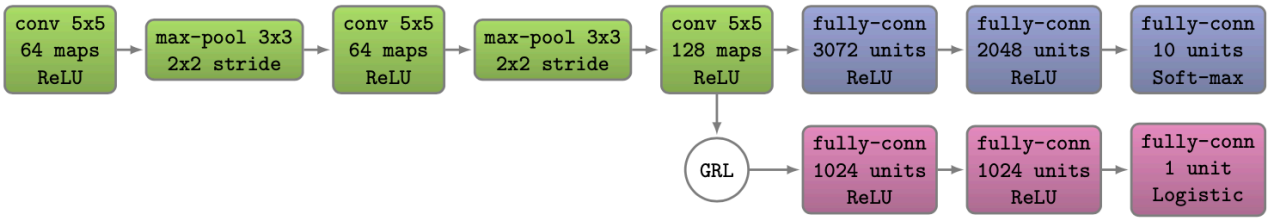
First time I made CNN this architecture (Pic 1). I used Adam optimiser and Xavier Initialization.



Picture 1

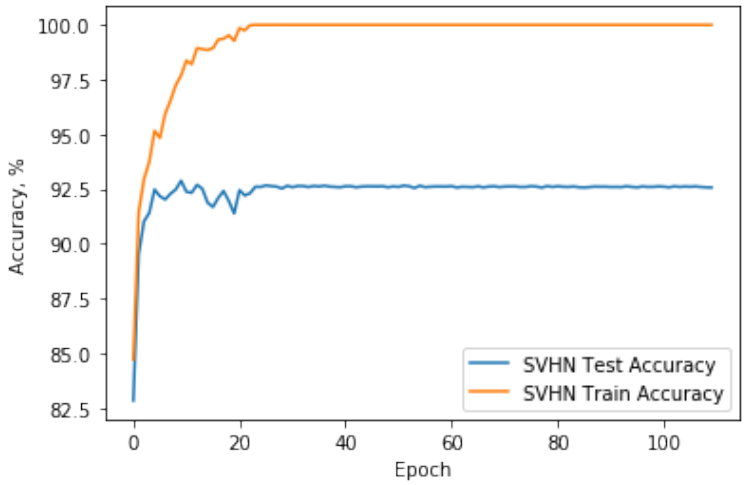
But I did not get good result (special in comparing with Baseline).

Then I change CNN on more complicated architecture (Pic 2). I have changed number of layers. In Domain Classifier from 100 to 1024 and in Label Predictor from 100 to 3072.



Picture 2

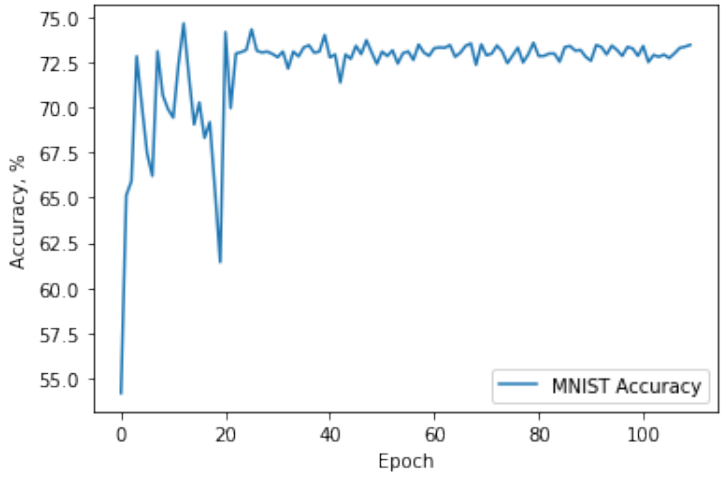
With this CNN I got accuracy on SVHN train and SVHN test (Graph 1):



Graph 1

As we can see the model gives really goon result (more than *99 %* accuracy) on SVHN train images, because the model was learned on this images. I get *92,5 %* accuracy on SVHN test images.

Accuracy on MNIST test set (Graph 2):

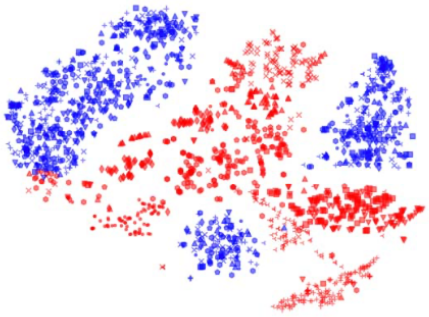


Graph 2

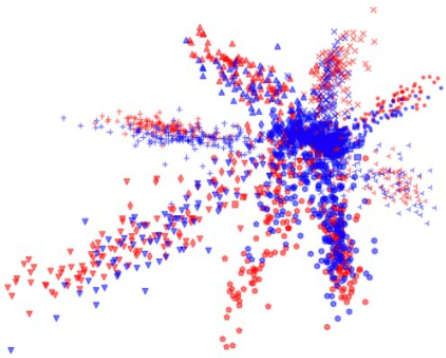
The model gives about *75 %* accuracy.

Latent space:

There is the distribution of source and target domain samples before (Pic 3) and after (Pic 4) the adaptation. The source (SVHN) and target domain (MNIST) samples are shown in red and blue respectively.



Picture 3 - Before adaptation



Picture 4 - After adaptation

Conclusion

In this course project I implemented Domain-Adversarial Training of Neural Networks. The adaptation is achieved through aligning the distortions of features across the two domains. The alignment is accomplished through standard backpropagation training.

I have learned how to build Neural Network, how to implement different optimisers, initialisations, how to use hyper-parameters and I learned what Domain Adaptation is.

My result is only *75 %* accuracy. From my point of view this is not excellent result. Maybe in order to improve I should also add Encoder which will embed input images from both domains to latent representation.

As a good practice would be make another method for example Duplex Generative Adversarial Network.

Application

Input:
— samples $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ and $T = \{\mathbf{x}_i\}_{i=1}^{n'}$,
— hidden layer size D,
— adaptation parameter λ,
— learning rate μ,
Output: neural network $\{\mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c}\}$

$\mathbf{W}, \mathbf{V} \leftarrow \text{random\_init}(\, D\,)$
$\mathbf{b}, \mathbf{c}, \mathbf{u}, d \leftarrow 0$
while stopping criterion is not met **do**
  for i from 1 to n **do**
    `# Forward propagation`
    $G_f(\mathbf{x}_i) \leftarrow \text{sigm}(\mathbf{b} + \mathbf{W}\mathbf{x}_i)$
    $G_y(G_f(\mathbf{x}_i)) \leftarrow \text{softmax}(\mathbf{c} + \mathbf{V}G_f(\mathbf{x}_i))$
    `# Backpropagation`
    $\Delta_{\mathbf{c}} \leftarrow -(\mathbf{e}(y_i) - G_y(G_f(\mathbf{x}_i)))$
    $\Delta_{\mathbf{V}} \leftarrow \Delta_{\mathbf{c}}\, G_f(\mathbf{x}_i)^\top$
    $\Delta_{\mathbf{b}} \leftarrow \left(\mathbf{V}^\top \Delta_{\mathbf{c}}\right) \odot G_f(\mathbf{x}_i) \odot (1 - G_f(\mathbf{x}_i))$
    $\Delta_{\mathbf{W}} \leftarrow \Delta_{\mathbf{b}} \cdot (\mathbf{x}_i)^\top$
    `# Domain adaptation regularizer...`
    `# ...from current domain`
    $G_d(G_f(\mathbf{x}_i)) \leftarrow \text{sigm}(d + \mathbf{u}^\top G_f(\mathbf{x}_i))$
    $\Delta_d \leftarrow \lambda(1 - G_d(G_f(\mathbf{x}_i)))$
    $\Delta_{\mathbf{u}} \leftarrow \lambda(1 - G_d(G_f(\mathbf{x}_i)))G_f(\mathbf{x}_i)$
    $\text{tmp} \leftarrow \lambda(1 - G_d(G_f(\mathbf{x}_i))) \times \mathbf{u} \odot G_f(\mathbf{x}_i) \odot (1 - G_f(\mathbf{x}_i))$
    $\Delta_{\mathbf{b}} \leftarrow \Delta_{\mathbf{b}} + \text{tmp}$
    $\Delta_{\mathbf{W}} \leftarrow \Delta_{\mathbf{W}} + \text{tmp} \cdot (\mathbf{x}_i)^\top$
    `# ...from other domain`
    $j \leftarrow \text{uniform\_integer}(1, \ldots, n')$
    $G_f(\mathbf{x}_j) \leftarrow \text{sigm}(\mathbf{b} + \mathbf{W}\mathbf{x}_j)$
    $G_d(G_f(\mathbf{x}_j)) \leftarrow \text{sigm}(d + \mathbf{u}^\top G_f(\mathbf{x}_j))$
    $\Delta_d \leftarrow \Delta_d - \lambda G_d(G_f(\mathbf{x}_j))$
    $\Delta_{\mathbf{u}} \leftarrow \Delta_{\mathbf{u}} - \lambda G_d(G_f(\mathbf{x}_j))G_f(\mathbf{x}_j)$
    $\text{tmp} \leftarrow -\lambda G_d(G_f(\mathbf{x}_j)) \times \mathbf{u} \odot G_f(\mathbf{x}_j) \odot (1 - G_f(\mathbf{x}_j))$
    $\Delta_{\mathbf{b}} \leftarrow \Delta_{\mathbf{b}} + \text{tmp}$
    $\Delta_{\mathbf{W}} \leftarrow \Delta_{\mathbf{W}} + \text{tmp} \cdot (\mathbf{x}_j)^\top$
    `# Update neural network parameters`
    $\mathbf{W} \leftarrow \mathbf{W} - \mu\Delta_{\mathbf{W}}$
    $\mathbf{V} \leftarrow \mathbf{V} - \mu\Delta_{\mathbf{V}}$
    $\mathbf{b} \leftarrow \mathbf{b} - \mu\Delta_{\mathbf{b}}$
    $\mathbf{c} \leftarrow \mathbf{c} - \mu\Delta_{\mathbf{c}}$
    `# Update domain classifier`
    $\mathbf{u} \leftarrow \mathbf{u} + \mu\Delta_{\mathbf{u}}$
    $d \leftarrow d + \mu\Delta_d$
  end for
end while

Algorithm - Stochastic training update