Neural Network Theory and Applications | Assignment#3

Shangyu Liu (020033910052)

May 5, 2021

1 Introduction

Individual differences in EEG signals lead to the poor generalization ability of EEG-based affective models. As introduced in the class, transfer learning can eliminate the subject differences and achieve appreciable improvement in recognition performance.

In this assignment, we built and evaluated a cross-subject affective model using Domain-Adversarial Neural Networks (DANN) with the SEED dataset.

A multi-layer fully-connected network (MLP) was adopted as the baseline model. Compared with the MLP model, DANN can effectively adapt to different domains and has a considerably better performance.

2 DANN

DANN is a domain adaptation approach with deep neural networks to eliminate the domain bias. The overview of its main structure is shown in Figure 1. The network can be generally divided into three modules, the feature extractor, the label predictor, and the domain classifier. The first layers works as a feature extractor which maps the input feature into a transformed high-dimensional neural space. Then the hidden features are passed to two different branches, the label predictor and the domain classifier, respectively. Exploiting the features, The label predictor classifies the sample into different class labels while the domain classifier distinguishes the domain source of the sample.

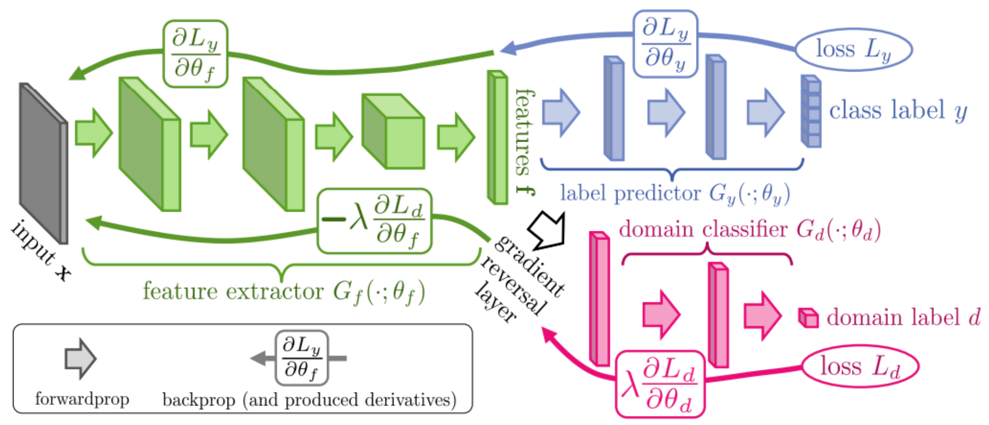
In terms of the process of forward and backward propagation, the extractor and the predictor together work like a normal multi-layer network. However, the gradients calculated by the domain classifier will be reversed before applying on the extractor. This means the domain classifier is learning to classify domains, but meanwhile cutting down the distinguishing ability of the extractor. This will lead to the hidden feature containing less domain-dependent information, which will also somehow affect the short-term performance of the label predictor. Therefore, these two branches are fighting against each other. In the end of training, we drop the domain classifier and combine the two remained modules to form one practical network, which is able to classify labels without domain preference.

The training process of each sample can be described concisely as follows:

(a) **Forward propagation.** The feature extractor $G_f(X, \theta_f)$ accepts the input X and inferences a hidden feature vector f. The label predictor $G_y(f, \theta_y)$ inferences the class label y and calculate the label loss L_y. Meanwhile, the domain classifier $G_d(f, \theta_d)$ inferences the domain label d and calculate the domain loss L_d.

(b) **Backward propagation.**

  i. **Compute the gradients.** The label predictor calculates the gradients $\frac{\partial L_y}{\partial \theta_y}$



$\frac{\partial L_y}{\partial \theta_f}$
$\frac{\partial L_y}{\partial \theta_y}$
loss L_y
class label y
features $\mathbf{f}$
label predictor $G_y(\cdot;\theta_y)$
input x
$-\lambda \frac{\partial L_d}{\partial \theta_f}$
gradient reversal layer
domain classifier $G_d(\cdot;\theta_d)$
feature extractor $G_f(\cdot;\theta_f)$
domain label d
$\lambda \frac{\partial L_d}{\partial \theta_d}$
loss L_d
forwardprop
backprop (and produced derivatives)

Figure 1: Overview of the Domain-Adversarial Neural Network [1]

and $\frac{\partial L_y}{\partial \theta_f}$. The domain classifier calculates the gradients $\frac{\partial L_d}{\partial \theta_d}$ and $\frac{\partial L_d}{\partial \theta_f}$.

ii. **Apply the gradients.** Update the parameters of G_y as

$$\theta'_y = \theta_y - \eta_y \frac{\partial L_y}{\partial \theta_y}$$

Update the parameters of G_d as

$$\theta'_d = \theta_d - \eta_d \frac{\partial L_d}{\partial \theta_d}$$

Update the parameters of G_f as

$$\theta'_f = \theta_f - (\eta_y \frac{\partial L_y}{\partial \theta_f} - \lambda \eta_d \frac{\partial L_d}{\partial \theta_f})$$

(c) **Check stop condition.** Check whether the iteration time exceeds the maximum epoch or the average decrement of the loss scores falls below the threshold. If so, the training process is over. Otherwise, continue with (a) and repeat the loop until condition satisfied.

In addition, it's essential to involve the data sampled from the testing domain during the training process, according to [1]. Therefore, some of the input data do not have class labels. While training them, we can skip the inference and back propagation process of the label predictor. For example, we update the parameters of G_f in step (b) as

$$\theta'_f = \theta_f + \lambda \eta_d \frac{\partial L_d}{\partial \theta_f}$$

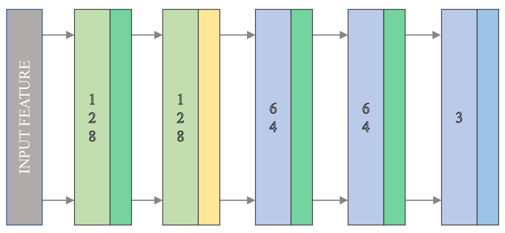
Moreover, we adopted a stochastic batch learning mode instead of online learning (update the parameters each time a single sample is fed). These details will be further discussed in the next section.

3 Implementation

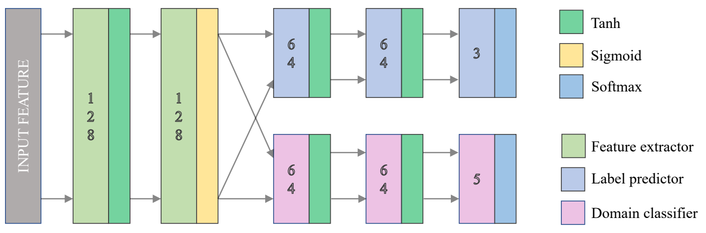
3.1 Overview

As shown in Figure 2(b), the structure of the implemented DANN follows a fixed empirical setting. The feature extractor has 2 layers, both with node number of 128. The label predictor and domain classifier both have 3 layers, with 64 nodes in each of the first two layers. The node number of the output layer of the predictor and classifier are 3 and 5, respectively. The activation function of the hidden layers are all set to tanh. The outputs of the extractor, predictor and classifier are activated by sigmoid, softmax and softmax, respectively. We adopted a cross entropy loss function for both the predictor and the classifier. The class and domain labels are mapped into one-hot vector in advance.

We also implemented a multi-layer fully-



(a) Baseline MLP



(b) DANN

Figure 2: Structure of the implemented MLP and DANN

connected network (MLP), which has exactly the same structure as the combination of the extractor and predictor in DANN, as shown in Figure 2(a). We compared DANN model with this model to show the effective domain-independent classifying ability of DANN, which will be further discussed in the next section.

These two models are implemented with Python 3.7.9 and TensorFlow 1.15. They are trained and tested with the SEED dataset, on a MacBook Pro laptop. The program is written in object-orientation style and the source code can be found at GitHub [2].

To further improve the performance, we use individual learning rate for the training of label predictor and domain classifier. Practically, we choose the optimizer of the predictor to update the parameter of the extractor uniformly.

$$\begin{aligned}\theta_f' &= \theta_f - (\eta_y \frac{\partial L_y}{\partial \theta_f} - \lambda \eta_d \frac{\partial L_d}{\partial \theta_f}) \\ &= \theta_f - \eta_y (\frac{\partial L_y}{\partial \theta_f} - \lambda \frac{\eta_d}{\eta_y} \frac{\partial L_d}{\partial \theta_f}) \\ &= \theta_f - \eta_y (\frac{\partial L_y}{\partial \theta_f} - \lambda' \frac{\partial L_d}{\partial \theta_f})\end{aligned}$$

In order to make it easy to adjust the hyperparameters, we define $\lambda' = \lambda \frac{\eta_d}{\eta_y}$. Finally, according to the experimental results, we set $\eta_y = 5e-5$, $\eta_d = 1e-4$ and $\lambda' = 0.3$.

3.2 Challenges

We came across significant technical issues while implementing the models. In this subsection, we will introduce three main challenges and the adopted method to overcome them.

3.2.1 Weight Initialization

We found the upper bound performance of MLP is quite sensitive to the weight initialization function, especially the deviation of the initial distribution. When we use a built-in function to initialize the weight, we found the training accuracy could be hardly larger than 0.6, no matter what the learning rate is. And as for optimizer except Adam, the training won't even converge.

After long-time investigation and experiments, we found the default standard deviation 1 of the built-in normal distribution [3] is too large for this model. As shown in Figure 3, the choice of the deviation is a non-trivial hyperparameter for the training process. Finally, we choose $stddev = 0.1$.

Actually, as the mean value of the normal distribution is set to 0, when the deviation is large, the weights are more likely to be large. Then the gradients of the loss function as well as the update pace would be large. The model may fluctuates heavily and even can not converge. On the contrary, when the deviation is small, the update pace of gradients would be small, and the network may converge too slowly or converge to a local-minimum point. Therefore, a moderate deviation is essential.

According to [4], the weights and biases can

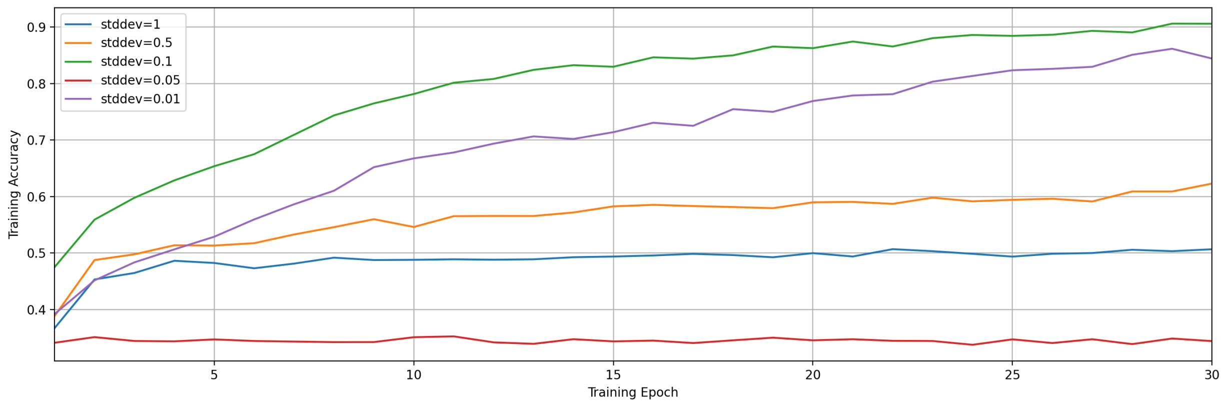


Figure 3: Training accuracy of MLP initialized by normal distribution with different standard deviation

be initialized as

$$W^l \sim N(\mu = 0, \sigma^2 = \frac{1}{n^{l-1}}), b^l = 0$$

We didn't follow this principal, instead just choose a moderate deviation empirically.

3.2.2 Gradients computation

TensorFlow 1.15 doesn't support fetching gradients from a execution session when there are None Values. The built-in function optimizer.compute_gradients will compute the gradient of the loss function with respect to all variables. However, some variables may not be included in the loss function, thus there exists tensors with None Value. If we run the gradient list with a session, we will get the error log, "TypeError: Fetch argument None has invalid type <class 'NoneType'>".

We solve this problem by decomposing the gradients based on the module boundary. The weights and biases of the feature extractor, label predictor, and domain classifier are added into different collections when initialized. Then we get all variables from each collection and specify the independent variables when computing the gradients. In this way, we eliminated None Values.

3.3 Batch-mode Learning

We implemented the training process following an on-line learning way and found the training speed unbearably slow. We optimized the operations to be run in a session and avoid redundant computation.

To further speed up the training process, we adopted a batch-mode learning method. In each epoch, the program randomly samples a batch of data from the training set and updates the parameters once all their gradients are computed. The process will repeat for times of total data number divided by the batch size.

Different from the training process of general models, there exists samples without class labels. A naive method to deal with this problem is to set the unknown class labels to zero vectors. As the labels are represented in one-hot vectors, this action will lead to the cross entropy loss set to zero. Then the parameters of the label predictor will not be updated. This method avoids the redundant condition commands, but unfortunately leads to the training accuracy decreased. The zero vectors will never be considered as a true prediction, as the output of the network are activated by softmax, always generating a unique label.

Instead, we defined different update operations for different input samples. The training process is then divided into two steps. We first

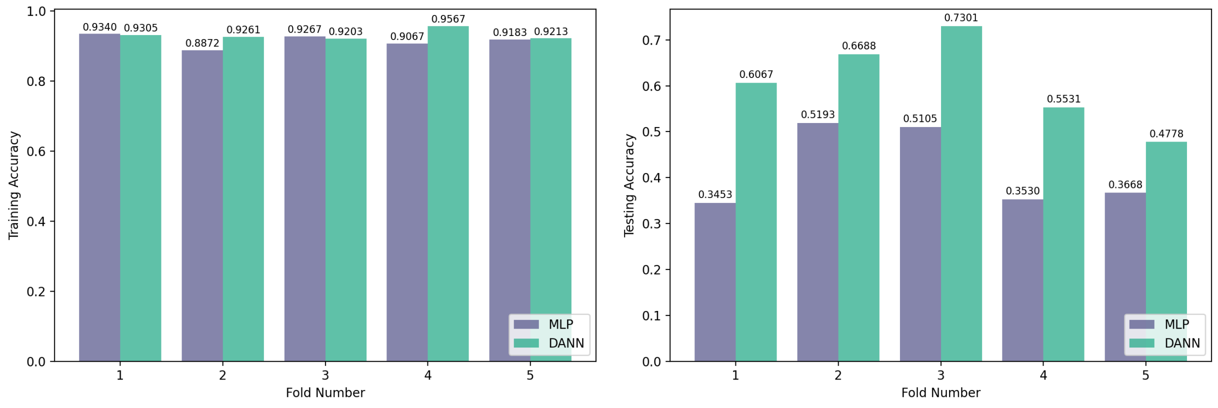


Figure 4: Comparison of Training and Testing Accuracy Between Baseline MLP and DANN

update the parameters of the extractor, predictor and classifier using fully-labeled data. Then we update the parameters of the extractor and classifier using samples without class labels. According to our experiments, putting these two steps together inside one batch iteration will lead to significantly better performance. The batch training loop are set to repeat for times of labeled data number divided by the batch size, in each epoch. As the number of labeled data are much larger than the unlabeled data, the unlabeled data are trained more effectively. The latter reflects domain knowledge of the testing samples, thus the testing performance will arise.

4 Experiments

To evaluate the performance of the DANN model, we performed a 5-folds cross-validation on the given SEED dataset. The 5 folds are divided exactly based on different subjects (i.e. the experiment participants while collecting the SEED dataset). Each time we leave one fold as the testing set and combine the others to form the training set.

Figure 4 shows the comparison of training and testing accuracy between baseline MLP and DANN. The accuracy score are evaluated after 60 epochs of training. Although the score fluctuates during time, we simply use the latest one to indicate the performance. We can see the training accuracy scores are very close while the testing accuracy is quite different. The DANN model defeats the baseline on every fold. Generally, the mean testing accuracy of MLP and DANN can be calculated as 0.4190 and 0.6073. The DANN model achieves nearly 20% higher accuracy. As a result, the DANN is experimentally proved to have fine generalization ability.

Figure 5 shows the training process of the DANN. We can see the accuracy of the domain classifier remains 0 during the training. The extractor is learning to extract domain-independent features from the input sample. After the training is complete, we drop the domain classifier and combine the feature extractor and the label predictor to perform inference.

5 Conclusion

In this assignment, we implemented a DANN model to perform domain-adaptive transfer learning in EEG-based affective scenario. The model is trained and tested on the SEED dataset. Compared to a baseline MLP with same structure volume, DANN achieves 20% higher testing accuracy. This experiment indicates that DANN can eliminate the subject differences and achieve appreciable improvement in recognition performance.

During the implementation of the model,

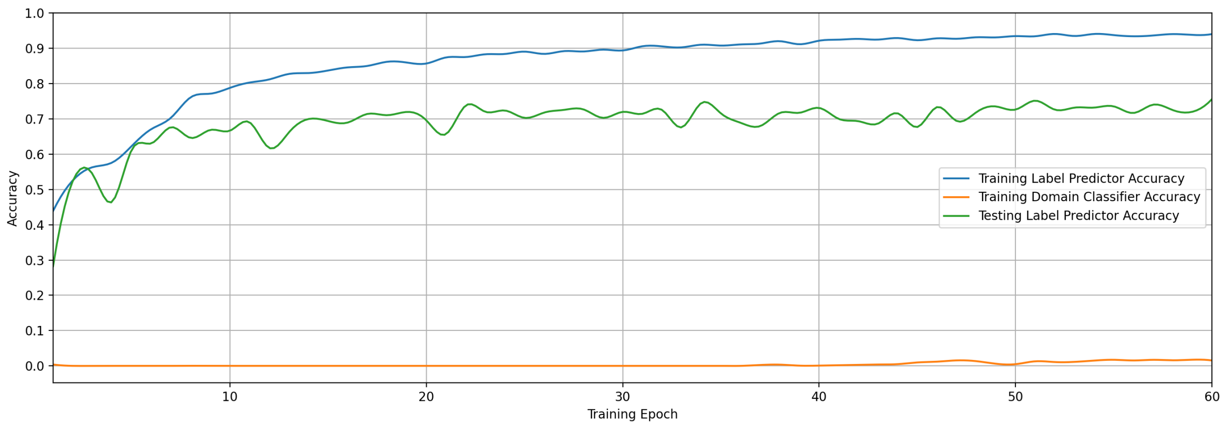


Figure 5: The Training Process of The DANN

we came across several non-trivial technical issues. We ran a deep investigation in these problems and gain useful knowledge and experience. The TensorFlow 1.x framework has been proved to have lots of deficiencies. Many failure can be avoided by using high-level backends, such as Keras. We need to learn more powerful tools which provide dynamic computation graph, such as TensorFlow 2.x and PyTorch. Meanwhile, we need to be aware of how to build up neural networks without advanced tools.

References

[1] Ganin, Y., Ustinova, E., Ajakan, H., Germain, P., Larochelle, H., Laviolette, F., ... & Lempitsky, V. (2016). Domain-adversarial training of neural networks. The journal of machine learning research, 17(1), 2096-2030.

[2] ThomasAtlantis:SJTUNNTA. `https://github.com/ThomasAtlantis/SJTUNNTA/blob/master/Assignment_3/DANN.py`

[3] tf.random.normal. `https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/random/normal`

[4] Glorot, X., & Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 249-256). JMLR Workshop and Conference Proceedings.