AI1603 Assignment Report: Recurrence and Exploration of Out-of-Distribution Algorithms

Xiangyuan Xue

Contents

1 Problem Setting

1.1 Out-of-Distribution

In common machine learning tasks, we tend to assume data obey I.I.D(Independently Identically Distribution). However, this assumption is hard to satisfy in realistic tasks. Imagine we have photos of cows and camels, and we are training a model(e.g. LeNet based on CNN) to make a binary classification. In our dataset, cows are always on the grass and camels are always in the desert, which makes our model believe the background is part of essential features of the animals. When cows appear in the desert and camels appear on the grass, the model fails disastrously in prediction.[1] In this case, there exists two completely opposite distribution in training set and test set, as is similar to our experiment dataset.

1.2 Colored MNIST Dataset

Colored MNIST is a modified MNIST dataset. $(0, 1, 2, 3, 4)$ and $(5, 6, 7, 8, 9)$ are respectively relabeled as 0 and 1. 60000 images of handwritten numeral are evenly divided into three groups: two training sets and one test set. 25% of labels are flipped to simulate realistic situation. Spurious correlation are created by coloring images red or green according to its label. Then these colors are flipped in different proportion according to groups.[1]

Environment	**Size**	**Label Flipped**	**Color Flipped**	**Theoretical Accuracy**
train1	20000	25%	20%	75%
train2	20000	25%	10%	75%
test	20000	25%	90%	75%

Table 1: Size and Distribution of Colored MNIST Dataset

Experiments will be carried out mainly on Colored MNIST. Multiple models and algorithms will be tried on this dataset, seeking for better performance in this Out-of-Distribution problem.

1.3 Intervention Analysis

To clearly express the intervention in Colored MNIST, we can draw graph below. Vertices represent features and edges represent correlations.

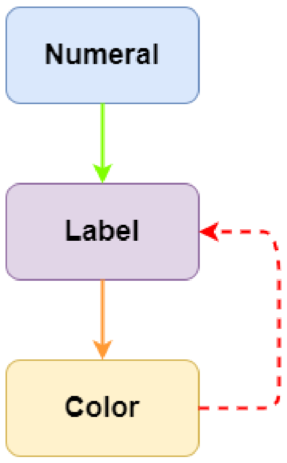


Figure 1: Intervention in Colored MNIST

Analysing the figure, we know the labels are directly and merely caused by numerals. Colors are artificial features caused by labels. However, our models do not distinguish between causes and effects. Then a **spurious correlation** will be made as the dotted-line arrow in the figure.

In general, numerals are called **causal feature**, and colors are called **non-causal feature**. And causal features are those which keep **invariant** across all the environments.

2 LeNet Implementation

2.1 LeNet Model

Our LeNet model here is basically the same with the one used for standard MNIST. But there is a little difference: the input images have 3 channels, instead of only 1. Moreover, it is a binary classification task, so the last linear layer has 2 outputs, instead of 10.

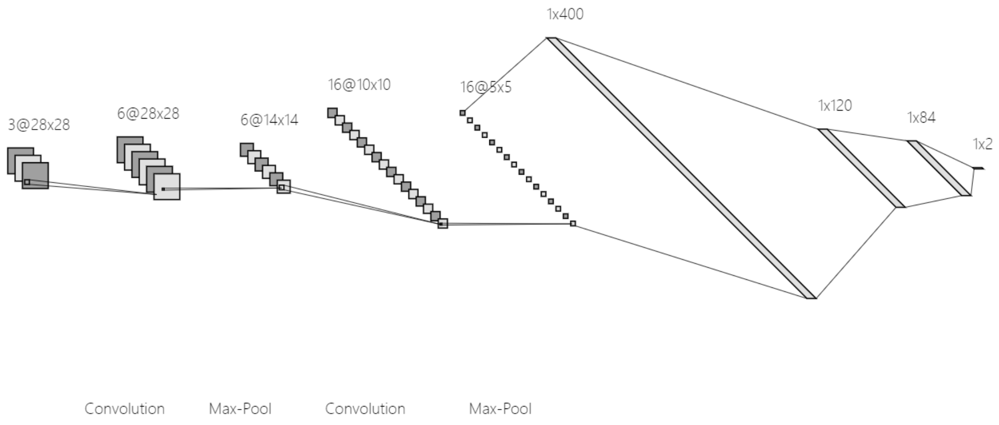
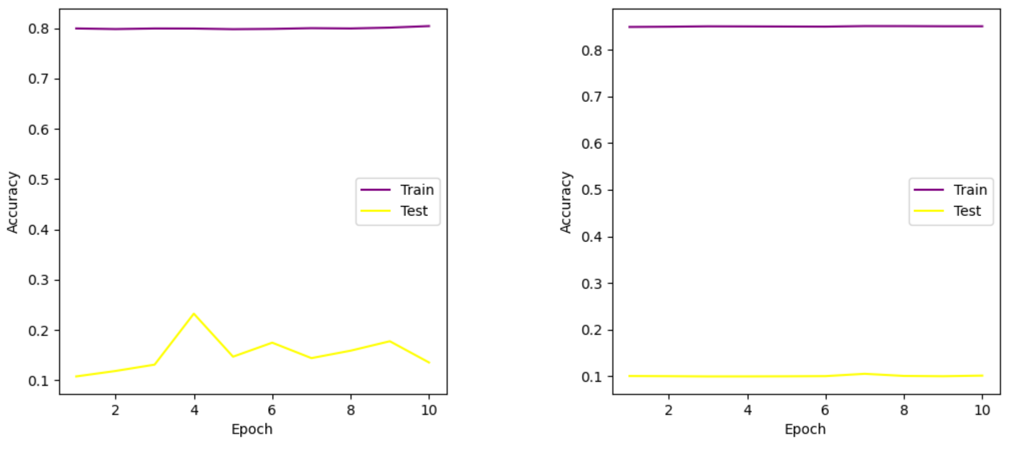


Figure 2: LeNet Model for Colored MNIST

2.2 Learning Curve

Actually, dataset *train1* and *train2* have similar distribution. But we have two separated training to thoroughly observe LeNet model's performance.



(a) Use *train1* Only (b) Both *train1* and *train2*

Figure 3: LeNet Learning Curve on Colored MNIST Dataset

First we train model on *train1* dataset only, test model on *test* dataset and plot its learning curve. Then we train model using both *train1* and *train2* dataset, test model on *test* dataset and plot its learning curve.

The results are quite disappointing. Although the model works well on training set, its performance on test set is terrible. Obviously, the model takes colors as a major feature, ignoring the shape of numerals to a great extent. Thus, it is cheated by flipped colors in test set.

2.3 Optimization

Traditional LeNet model fails disastrously in Colored MNIST dataset because of artificially constructed data. However, we still have some remedial measures to optimize its performance and improve its generalization ability. By pre-processing the data, such as data augmentation and enhancement, we can save the performance of LeNet model. Related works are completed by *Bangjun Wang* and can be seen in his report.

3 IRM Algorithm

3.1 Basic Ideology

IRM(Invariant Risk Minimization) is an algorithm proposed by Martin Arjovsky et al. in 2019, as opposed to ERM(Empirical Risk Minimization). Since ERM fails to distinguish causality from spurious correlation, IRM seeks for a classifier balancing among different environments, which is also called **invariant predictor**. However, finding an optimal predictor in all the environments is usually impossible. So some assumptions are given and constraints are transformed into penalty. By mathematical methods, invariant risk can be written as a convex loss function, which is simple enough for machine learning to solve.

3.2 Mathematical Derivation

Essence of IRM algorithm lies in mathematical derivation. Without profound comprehension of mathematical derivation, we cannot truly understand IRM algorithm, not to mention optimization. However, the symbol system in original paper is a little complex and the derivation lacks coherence. We organized the derivation process and added some details here.[1]

In a classification task, our target is to find a data representation $\boldsymbol{\Phi} : \boldsymbol{X} \to \boldsymbol{H}$ and a predictor $\boldsymbol{\omega} : \boldsymbol{H} \to \boldsymbol{Y}$ which are optimal in minimization risk:

$$\min_{\substack{\boldsymbol{\Phi}:\boldsymbol{X}\to\boldsymbol{H} \\ \boldsymbol{\omega}:\boldsymbol{H}\to\boldsymbol{Y}}} \sum_{e\in E_{\text{train}}} R^e(\boldsymbol{\omega}\circ\boldsymbol{\Phi}) \quad \text{s.t.} \quad \forall e\in E_{\text{train}} : \boldsymbol{\omega} \in \arg\min_{\boldsymbol{\omega}^*:\boldsymbol{H}\to\boldsymbol{Y}} R^e(\boldsymbol{\omega}^*\circ\boldsymbol{\Phi})$$

Bi-leveled optimization can be quite hard, so we **weaken constraints into penalty term**, and write down a loss function:

$$L(\boldsymbol{\Phi}, \boldsymbol{\omega}) = \sum_{e \in E_{\text{train}}} \underbrace{R^e(\boldsymbol{\omega} \circ \boldsymbol{\Phi})}_{\text{empirical risk}} + \underbrace{\lambda D(\boldsymbol{\Phi}, \boldsymbol{\omega}, e)}_{\text{invariance}}$$

* Notice: $\lambda \in [0, \infty)$ is a factor balancing empirical risk and invariance.

To further simplify the function, assume $\boldsymbol{\omega}$ is **linear** and consider single environment e, we have:

$$\boldsymbol{Y}^e = \boldsymbol{\omega} \circ \boldsymbol{\Phi}(\boldsymbol{X}^e) \quad \xRightarrow{\text{in matrix form}} \quad \boldsymbol{\Phi}(\boldsymbol{X}^e)\boldsymbol{\omega} = \boldsymbol{Y}^e$$

We can easily write down **least square solution**:

$$\boldsymbol{\omega}^e_{\boldsymbol{\Phi}} = \left[\boldsymbol{\Phi}(\boldsymbol{X}^e)^T \boldsymbol{\Phi}(\boldsymbol{X}^e)\right]^{-1} \boldsymbol{\Phi}(\boldsymbol{X}^e)^T \boldsymbol{Y}^e$$

Make some equivalent transformation:

$$\boldsymbol{\Phi}(\boldsymbol{X}^e)^T \boldsymbol{\Phi}(\boldsymbol{X}^e)\boldsymbol{\omega}^e_{\boldsymbol{\Phi}} - \boldsymbol{\Phi}(\boldsymbol{X}^e)^T \boldsymbol{Y}^e = \boldsymbol{0}$$

Then distance can be defined for optimization:

$$D(\boldsymbol{\Phi}, \boldsymbol{\omega}, e) = \|\boldsymbol{\Phi}(\boldsymbol{X}^e)^T \boldsymbol{\Phi}(\boldsymbol{X}^e)\boldsymbol{\omega} - \boldsymbol{\Phi}(\boldsymbol{X}^e)^T \boldsymbol{Y}^e\|^2$$

* Notice: consider $\left(k\boldsymbol{\Phi}, \frac{1}{k}\boldsymbol{\omega}\right)$ and $k \to 0$, we have $D(\boldsymbol{\Phi}, \boldsymbol{\omega}, e) = 0$, it is a disaster.

But there's another interesting thing:

$$\boldsymbol{\omega} \circ \boldsymbol{\Phi} = \underbrace{\left(\boldsymbol{\omega} \circ \boldsymbol{\varphi}^{-1}\right)}_{\boldsymbol{\omega}'} \circ \underbrace{(\boldsymbol{\varphi} \circ \boldsymbol{\Phi})}_{\boldsymbol{\Phi}'}$$

So we might as well fix $\boldsymbol{\omega} = \boldsymbol{\omega_0}$, then:

$$\begin{aligned}
L_{\boldsymbol{\omega_0}} &= \sum_{e \in E_{\text{train}}} R^e(\boldsymbol{\omega_0} \circ \boldsymbol{\Phi}) + \lambda D_{\boldsymbol{\omega_0}}(\boldsymbol{\Phi}, e) \\
&= \sum_{e \in E_{\text{train}}} R^e(\boldsymbol{\omega_0} \circ \boldsymbol{\Phi}) + \lambda \|\boldsymbol{\Phi}(\boldsymbol{X}^e)^T \boldsymbol{\Phi}(\boldsymbol{X}^e)\boldsymbol{\omega} - \boldsymbol{\Phi}(\boldsymbol{X}^e)^T \boldsymbol{Y}^e\|^2 \\
&= \sum_{e \in E_{\text{train}}} R^e(\boldsymbol{\omega_0} \circ \boldsymbol{\Phi}) + \lambda \frac{\partial}{\partial \boldsymbol{\omega}} \left[\frac{1}{2} \left(\boldsymbol{\Phi}(\boldsymbol{X}^e)\boldsymbol{\omega} - \boldsymbol{Y}\right)^T \left(\boldsymbol{\Phi}(\boldsymbol{X}^e)\boldsymbol{\omega} - \boldsymbol{Y}\right)\right]_{\boldsymbol{\omega_0}} \\
&= \sum_{e \in E_{\text{train}}} R^e(\boldsymbol{\omega_0} \circ \boldsymbol{\Phi}) + \lambda \|\nabla_{\boldsymbol{\omega}} R^e(\boldsymbol{\omega} \circ \boldsymbol{\Phi})\|^2_{\boldsymbol{\omega_0}}
\end{aligned}$$

* Part of the mathematical derivation is referenced from notes of Yearn online.[5]

The equation above reveals the ultimate goal of IRM in an amazingly simplified form. The recurring term $R^e(\boldsymbol{\omega} \circ \boldsymbol{\Phi})$ hints that empirical risk and invariance penalty are isomorphic in mathematical expressions. Once we calculate empirical risk, we can effortlessly figure out invariance penalty by calculating the gradient of empirical risk to linear predictor $\boldsymbol{\omega}$. This feature makes the loss function easy to calculate and greatly reduced coding difficulty.

It is also necessary to note that the specific form of R^e does not matter. In classification tasks we can use cross-entropy loss function, and in regression tasks we can use mean-square-error loss function.

3.3 Algorithm Implementation

Problem Resolution

Derivation above shows mathematical form of IRM. If we want to implement the algorithm in *PyTorch*, there are still some details.

Input images have the shape of $(3, 28, 28)$, containing 2352 components. If we build an MLP model, at least 4 hidden layers are needed, which gives the model great complexity and makes training process quite slow. Imitating the implementation in original paper, we resize the images from 28×28 to 14×14, containing 588 components(only a quarter of original images). What's more, we only need 3 hidden layers to ensure MLP model works well. This transformation lower the experiment cost to a great extent.

Then a fatal problem occurs. When we calculate the gradient term using *grad* function in *torch.autograd*, we multiply the training loss $R^e(\boldsymbol{\omega} \circ \boldsymbol{\Phi})$ by $\boldsymbol{\omega}$ and get the gradient. However, in this model, the data representation $\boldsymbol{\Phi}$ is no longer a naive linear function. We only get gradient of the last layer, ignoring all the layers before.

Let we just put this problem aside and add some derivation. Consider $\boldsymbol{\Phi}$ as a compound function:

$$\boldsymbol{\Phi} = f \circ g$$

Then rewrite the invariance penalty term:

$$\begin{aligned} P &= \|\nabla_{\boldsymbol{\omega}} R^e(\boldsymbol{\omega} \circ \boldsymbol{\Phi})\|^2_{\boldsymbol{\omega_0}} \\ &= \|\nabla_{\boldsymbol{\omega}} R^e(\boldsymbol{\omega} \circ f \circ g)\|^2_{\boldsymbol{\omega_0}} \\ &= \|\nabla_{\boldsymbol{\omega}} R^e(\boldsymbol{\omega} \circ f_g)\|^2_{\boldsymbol{\omega_0}} + \underbrace{\|\nabla_{\boldsymbol{\omega}} R^e(\boldsymbol{\omega} \circ g)\|^2_{\boldsymbol{\omega_0}}}_{\text{ignored term}} \end{aligned}$$

Coincidentally, the ignore term is composed by all the parameters in former layers. For an MLP model, the output can be expressed by a complex compound function $F = f_1 \circ f_2 \circ \cdots \circ f_n$

where $f_i \in F$ is linear function $f_i = k_{i,1}x_1 + k_{i,2}x_2 + \cdots + k_{1,m}x_m$, then the ignored term can be written as:

$$\|\nabla_{\boldsymbol{\omega}} R^e(\boldsymbol{\omega} \circ g)\|^2_{\boldsymbol{\omega_0}} = \mu \sum_{f_i \in F} \sum_{k_j \in f} k^2_{i,j}$$

* μ is the term which we already figure out.

It seems that the problem is perfectly solved. But in practice, it is a little tricky because all the layers are attached by *ReLU* activation function, making functions non-linear. A solution can be found in code repository for IRM on *GitHub*(Although it is strange that nothing about this is mentioned in original paper).[6] In the repository, μ is no longer the result of *autograd* function. It is treated as a hyper-parameter. By hundreds of adjustment, $\mu \approx 0.001$ is confirmed suitable.

MLP Model

As previous paragraph introduced, MLP model for our IRM implementation is simple, containing 1 input layer, 3 hidden layer and 1 output layer. The output layer has only 1 component to predict the classification. The prediction depends on whether this component is positive or negative.

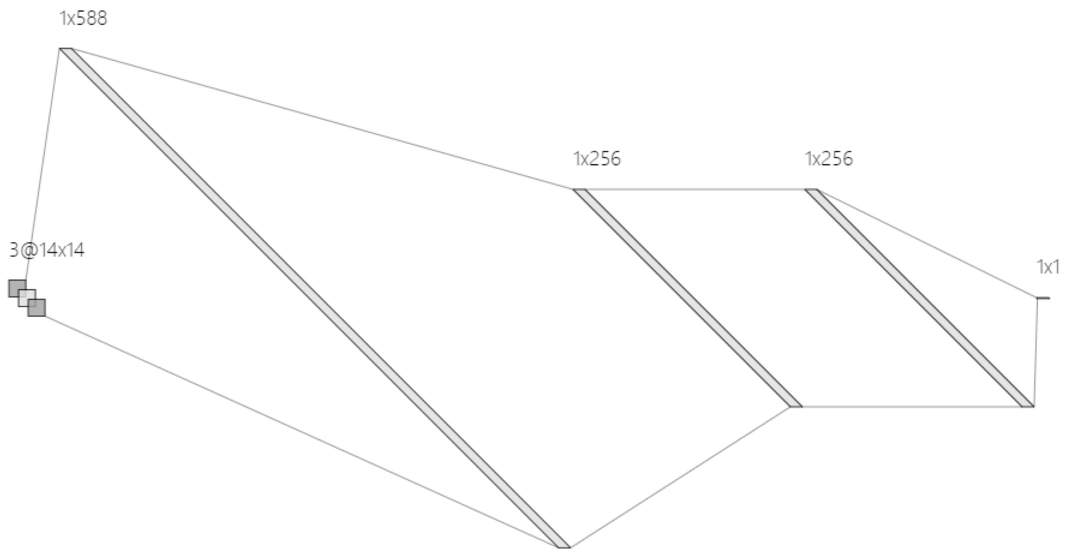


Figure 4: MLP Model for IRM Implementation

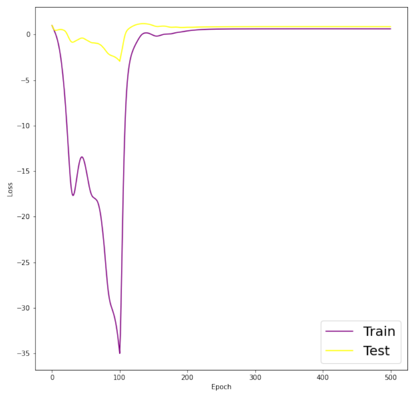
Learning Curve

It is necessary to note that there is another hyper-parameter λ, which is also called penalty weight in coding. Here we just follow the code repository and let penalty weight be a function of epoch:

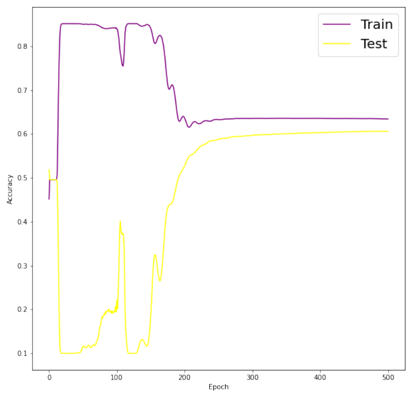
$$\text{penalty weight} = \begin{cases} 1.0 & \text{epoch} \leq 100 \\ 10000.0 & \text{epoch} > 100 \end{cases}$$

* The reasons and analysis will be shown later.

In this experiment, we put two environments *train1* and *train2* together. IRM algorithm will automatically eliminate the difference between two environments. Loss and accuracy are defined as mean of the two sets. We plot learning curve after training the model.



(a) Loss Curve



(b) Accuracy Curve

Figure 5: IRM on MLP Model Learning Curve on Colored MNIST Dataset

We can analyse the learning behavior of IRM algorithm through the learning curve. On training set, loss of the model drops smoothly in first 100 epochs. At the same time, accuracy rises quickly in first 20 epochs and remains stable. But on the test set, loss drops slowly and accuracy remains at a low level. After 100 epochs, invariance penalty is given an extremely high weight. The model quickly adjusts itself to reduce invariance penalty. Accuracy on training set drops, while it rises on test set. After 300 epochs, accuracy of both sets converges are remain at a fixed level.

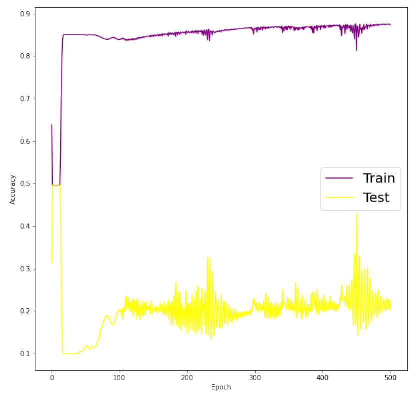
The results show that IRM algorithm has an splendid performance on Colored MNIST dataset, achieving approximately 65% accuracy with a relatively low time and space cost. IRM entirely defeats ERM in OOD problems where spurious correlation exists.

3.4 Adaptive Improvement

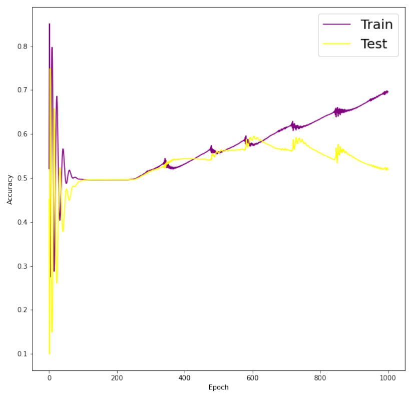
Sensitiveness to Penalty Weight

There is a unique hyper-parameter we have not formally talked about. That is penalty weight λ. According to the original paper, it depends on the specific problem we face and the model we choose. In synthetic data experiment, $\lambda = 10^5$ is a fixed value(This is reflected only in coding).[1] It turns out that this penalty weight works well.

When it comes to Colored MNIST, things are totally different. We respectively choose a tiny penalty weight $\lambda = 1.0$ and a relatively large penalty weight $\lambda = 10000.0$ to see the learning curve.



(a) Penalty Weight $\lambda = 1.0$



(b) Penalty Weight $\lambda = 10000.0$

Figure 6: IRM with Constant Penalty Weight Learning Curve

Tiny penalty weight $\lambda = 1.0$ means the invariance term $D_{\omega_0}(\mathbf{\Phi}, e)$ is of little importance. Thus, the model constantly tries to minimize the empirical risk term $R^e(\boldsymbol{\omega_0} \circ \mathbf{\Phi})$, leading to high accuracy on training set but low accuracy on test set. Large penalty weight over-emphasizes the importance of the invariance term $D_{\omega_0}(\mathbf{\Phi}, e)$, making the learning process hesitant. Thus, convergence speed slows down and the final accuracy is lower than expectation.

The discussion above reveals IRM algorithm's **sensitiveness** to penalty weight λ. On one hand, low penalty weight denies the importance of invariance, resulting in low accuracy on test set. On the other hand, high penalty weight limits the minimization of empirical risk, lowering convergence speed and final accuracy.

Self-Adaptive Optimization

Since we know that constant penalty weight λ does not perform well on Colored MNIST dataset, finding an alternative solution becomes necessary. We are going to improve the selection of penalty weight λ, enhancing robustness of IRM algorithm.

Let we observe the specific form of IRM loss function here:

$$L = \mu \sum_{f_i \in F} \sum_{k_j \in f} k_{i,j}^2 + \sum_{e \in E_{\text{train}}} R^e(F) + \lambda \|\nabla_{\boldsymbol{\omega}} R^e(\omega f_n)\|_{\omega=1.0}^2$$

A reasonable idea is to dynamically adjust penalty weight λ, ensuring empirical risk and invariance penalty are in the same order of magnitude. So make equivalent transformation:

$$\lambda = \frac{L - \mu \sum_{f_i \in F} \sum_{k_j \in f} k_{i,j}^2 - \sum_{e \in E_{\text{train}}} R^e(F)}{\|\nabla_{\boldsymbol{\omega}} R^e(\omega f_n)\|_{\omega=1.0}^2}$$

Observing the expression, we notice that the denominator is the penalty directly calculated by *grad* function, and the numerator is actually the loss of training(without invariance penalty). So we can conclude a simple proportional relationship:

$$\lambda \sim \frac{L_{\text{Train}}}{P_{\text{Train}}}$$

* L represents loss and P represents penalty.

Another problem is the speed of convergence. It is already known that penalty weight λ decides the focus of training, and the reduction of loss on training set is much faster than that on test set. Then we can understand the given optimization in code repository.[6] Penalty weight λ is first assigned a tiny value to promote the reduction of empirical risk. When the loss on training set is about to converge, penalty weight λ is switched to a large value, forcing the model to reduce the invariance penalty. Therefore, we effectively accelerate the training process.

Now we can write down the formula of **Self-Adaptive Optimized** penalty weight λ:

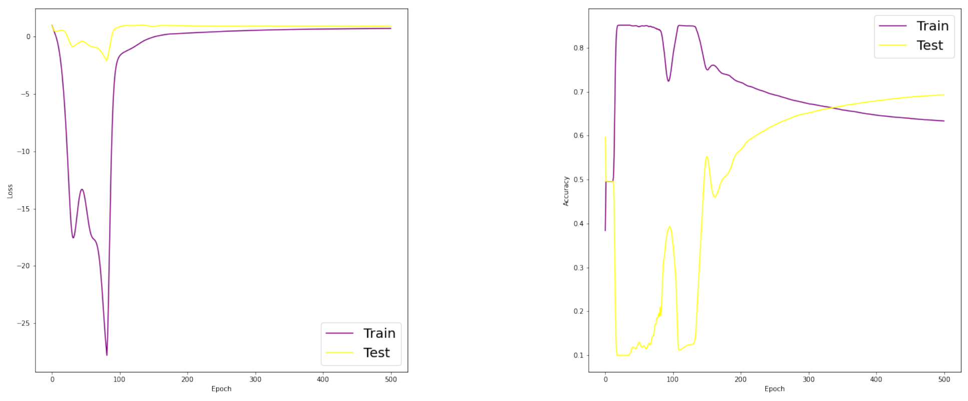
$$\lambda = \begin{cases} \epsilon & t \leq \tau \\ \xi \dfrac{L_{\text{Train}}}{P_{\text{Train}}} & t > \tau \end{cases}$$

* t represents training step, and τ is when the model converge in the first stage.

* ϵ is a value tiny enough to allow reduction of empirical risk.

* ξ is a coefficient which doesn't matter in most cases.

Then we apply **Self-Adaptive Optimization** to IRM algorithm and plot the learning curve. Its performance is amazingly excellent.



(a) Loss Curve (b) Accuracy Curve

Figure 7: IRM with Self-Adaptive Optimization Learning Curve

Compared to original IRM implementation, IRM with **Self-Adaptive Optimization** converge at a similar speed and end up with a higher accuracy on test set. What is revolutionary is that penalty weight λ no longer relies on manual adjustment. It adjusts itself to optimize the model's performance.

Algorithm	**Average Accuracy**	**Self-Adaptive**
IRM in original paper	60.93%	No
IRM with **Self-Adaptive Optimization**	64.79%	Yes

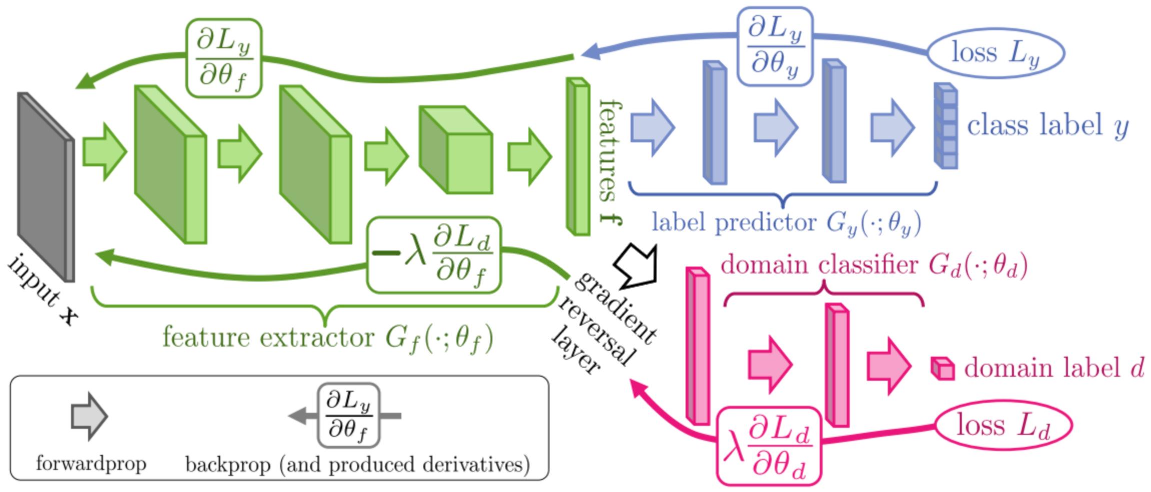
Table 2: Performance Enhancement by **Self-Adaptive Optimization**

Comparison of performance between the models with and without **Self-Adaptive Optimization** is presented in the table above.

4 DANN Algorithm

4.1 Brief Overview

DANN(Domain-Adversarial Training of Neural Networks) is a famous **Domain Adaptation** algorithm in **Transfer Learning** proposed by Yaroslav Ganin et al. in 2016. It is inspired by GAN(Generative Adversarial Network). In GAN, a generator is constructed to generate fake data and strengthen the robustness of discriminator.[3]



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

Figure 8: DANN Model in Original Paper

Correspondingly, there is a feature extractor θ_f, a label predictor θ_y, and a domain classifier θ_d in DANN model. The feature extractor θ_f transforms input data into an eigenvector $f \in \mathbb{R}^k$. The domain classifier performs a binary classification task where 0 stands for source domain and 1 stands for target domain. Similar to the ideology of IRM, we are going to minimize the error in label predictor and maximize the error in domain classifier.

Then loss function can be given:

$$L(\theta_f, \theta_y, \theta_d) = \sum_{i=1}^{n} L_y^i(\theta_f, \theta_y)|_{d_i=0} - \lambda \sum_{i=1}^{n} L_d^i(\theta_f, \theta_d)$$

Consider the gradient update of θ_f:

$$\theta_f \leftarrow \theta_f - \mu \left(\frac{\partial L_y^i}{\partial \theta_f} - \lambda \frac{\partial L_d^i}{\partial \theta_f} \right)$$

We can see the gradient of label predictor and domain classifier are contrary to each other. This will cause a lot of trouble when coding, so **GRL**(Gradient Reversal Layer) is proposed. It can be viewed as a function F satisfying:

$$\begin{cases} F(x) = x \\ \dfrac{\mathrm{d}F}{\mathrm{d}x} = -k_p \end{cases}$$

* $k_p = \frac{2}{1+e^{-\gamma p}} - 1$ changes with the process of training(see paper for details).

GRL brings convenience because it makes coding easy and realizes end-to-end training process. It is the essence of DANN.

Brief overview above is referenced from notes of Puwei Dai online.[4]

4.2 Experiment Application

To apply DANN to Colored MNIST dataset, we can take *train1* and *train2* as source domain, and take *test* as target domain. By transfer learning from training set to test set, DANN is supposed to find out the causal feature and have a relatively satisfying performance.

Related works are completed by *Bangjun Wang* and can be seen in his report.

5 Algorithm Exploration

5.1 Algorithm Proposition

OOD algorithms mentioned before are all based on supervised learning. Thus, They inevitably focus on the correlation between features and labels, and are finally more or less affected by spurious correlation. So how about unsupervised learning? It is valuable to make some attempts.

Now our goal is to design a model supporting unsupervised learning and extracting common features in training and test set. Inspired by AutoEncoder, a symmetric MLP model is built.

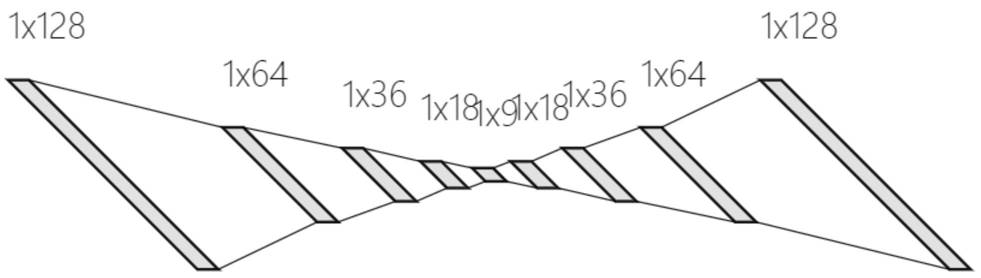


Figure 9: MLP Model for Feature Extracting

Similar to AutoEncoder, we call the left side **encoder**, and call the right side **decoder**. The vector in the middle with 9 components is the feature that we extract. Different from AutoEncoder, we try to extract common features in training and test set, so both of them will be the input of this model.

After training the model, we compress all the data from training and test set into feature vectors using encoder of the model. Then conventional machine learning algorithms can be used on them. Here we use *RandomForestClassifier* in *scikit-learn* which has sufficient representation ability to finish this task.

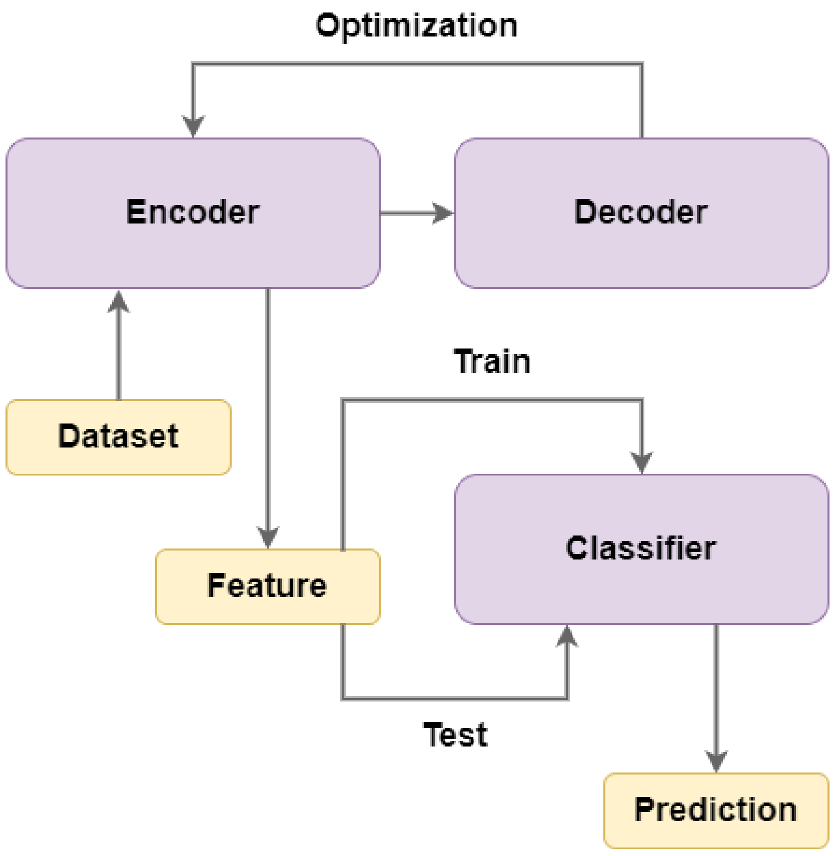


Figure 10: Transfer Learning with Unsupervised Dimension Reduction

5.2 Convergence Analysis

Suppose we have encoder ϕ and decoder φ, then feature vector $\boldsymbol{\alpha} = \phi(\boldsymbol{X})$, and mirror $\boldsymbol{X'} = \varphi(\boldsymbol{\alpha}) = \varphi(\phi(\boldsymbol{X}))$, we are looking for optimal ϕ and φ, resulting in least difference between $\boldsymbol{X}$ and $\boldsymbol{X'}$ on both training and test set.

Here we define the loss function:

$$L(\phi, \varphi) = R(\boldsymbol{X'}, \boldsymbol{X}) = R(\varphi(\phi(\boldsymbol{X})), \boldsymbol{X})$$

For convenience we define $\boldsymbol{\Phi} = \varphi \circ \phi$, so

$$L(\boldsymbol{\Phi}) = R(\boldsymbol{\Phi}(\boldsymbol{X}), X)$$

This function is applied to both training and test set:

$$\boldsymbol{X'}_{\text{train}} = \boldsymbol{\Phi}(\boldsymbol{X}_{\text{train}})$$
$$\boldsymbol{X'}_{\text{test}} = \boldsymbol{\Phi}(\boldsymbol{X}_{\text{test}})$$

Loss in training and test set respectively accounts for a certain proportion:

$$L(\boldsymbol{\Phi}) = L_{\text{train}}(\boldsymbol{\Phi}) + \lambda L_{\text{test}}(\boldsymbol{\Phi})$$

* $\lambda \in [0, +\infty)$ depends on the ratio of images numbers in training and test set.

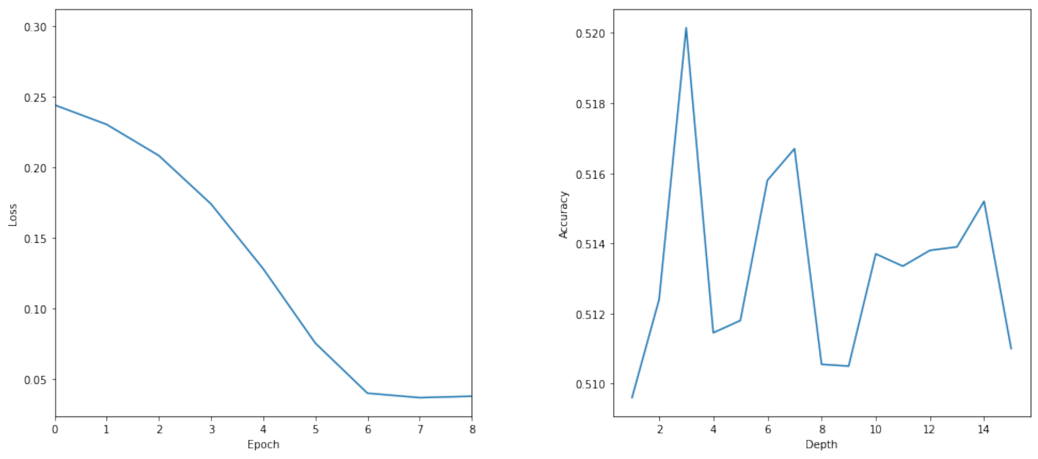
Simplify the expression:

$$\begin{aligned}
L(\boldsymbol{\Phi}) &= R(\boldsymbol{X'}_{\text{train}}, \boldsymbol{X}_{\text{train}}) + \lambda R(\boldsymbol{X'}_{\text{test}}, \boldsymbol{X}_{\text{test}}) \\
&= R(\boldsymbol{\Phi}(\boldsymbol{X}_{\text{train}}), \boldsymbol{X}_{\text{train}}) + \lambda R(\boldsymbol{\Phi}(\boldsymbol{X}_{\text{test}}), \boldsymbol{X}_{\text{test}}) \\
&= [\boldsymbol{\Phi}(\boldsymbol{X}_{\text{train}}) - \boldsymbol{X}_{\text{train}}]^T [\boldsymbol{\Phi}(\boldsymbol{X}_{\text{train}}) - \boldsymbol{X}_{\text{train}}] \\
&\quad + \lambda [\boldsymbol{\Phi}(\boldsymbol{X}_{\text{test}}) - \boldsymbol{X}_{\text{test}}]^T [\boldsymbol{\Phi}(\boldsymbol{X}_{\text{test}}) - \boldsymbol{X}_{\text{test}}] \\
&= \left[\boldsymbol{\Phi}\begin{pmatrix} \boldsymbol{X}_{\text{train}} \\ \sqrt{\lambda}\boldsymbol{X}_{\text{test}} \end{pmatrix} - \begin{pmatrix} \boldsymbol{X}_{\text{train}} \\ \sqrt{\lambda}\boldsymbol{X}_{\text{test}} \end{pmatrix}\right]^T \left[\boldsymbol{\Phi}\begin{pmatrix} \boldsymbol{X}_{\text{train}} \\ \sqrt{\lambda}\boldsymbol{X}_{\text{test}} \end{pmatrix} - \begin{pmatrix} \boldsymbol{X}_{\text{train}} \\ \sqrt{\lambda}\boldsymbol{X}_{\text{test}} \end{pmatrix}\right] \\
&= R\left(\boldsymbol{\Phi}\begin{pmatrix} \boldsymbol{X}_{\text{train}} \\ \sqrt{\lambda}\boldsymbol{X}_{\text{test}} \end{pmatrix}, \begin{pmatrix} \boldsymbol{X}_{\text{train}} \\ \sqrt{\lambda}\boldsymbol{X}_{\text{test}} \end{pmatrix}\right)
\end{aligned}$$

Actually, this loss function is nothing different with conventional loss function, though its variable seems a little scary. So its convergence is proved and convex optimization is available.

5.3 Experiment Application

We apply unsupervised dimensional reduction to classification task on Colored MNIST dataset. *train1* and *train2* are mixed up as training set. Both training and test set are put into MLP model for training. After that, all the data are transformed into feature vectors using encoder in the model. Then these features are used to train the random forest classifier. Loss curve in MLP model training and accuracy curve with depth of random forest are plotted.



(a) Loss Curve (b) Accuracy Curve

Figure 11: Learning Curve with Unsupervised Dimension Reduction

Figure shows that loss in MLP model training converges fast. After feature extracting, random forest classifier has approximately 51% accuracy on test set and the highest accuracy reaches 52%, which is slightly better than random guessing and DANN without optimization. Thus, it is a relatively effective OOD algorithm on Colored MNIST dataset.

6 Conclusion

6.1 Contribution

All my contributions have been presented in this report. Firstly, I built a naive LeNet as a baseline, finding disastrous performance of traditional ERM algorithms. Secondly, I read the IRM paper and researched mathematical principles behind the algorithm. I sorted out the derivation of IRM and make up some details. Then I researched the implementation of IRM, building MLP model and solving several problems for coding, including image transformation and gradient calculation. I also researched the performance of IRM algorithm under different penalty weight and looked for improvements by dynamically adjusting penalty weight. Moreover, I read the DANN paper and described its mathematical principles. Finally, I proposed an OOD algorithm based on unsupervised feature extracting and analysed its convergence. The algorithm was proved effective on Colored MNIST dataset.

6.2 Innovation

In the implementation of IRM algorithm, I researched the sensitiveness to penalty weight and proposed self-adaptive optimization for penalty weight. This optimization avoids manual adjustment for hyper-parameter and to some extent improves the performance of IRM algorithm.

In my exploration process, I proposed an OOD algorithm based on unsupervised feature extracting and analysed its convergence. It has a relatively satisfying performance on Colored MNIST dataset. Unsupervised learning is likely to eliminate spurious correlation and is a promising research direction in domain adaptation.

6.3 Code Repository

All the source code and figures are shared on my GitHub. Recurrence of all the experiment results can be realized through the project.

References

[1] Martin Arjovsky, Léon Bottou, Ishaan Gulrajani, and David Lopez-Paz. Invariant Risk Minimization. arXiv:1907.02893, 2019.

[2] Elan Rosenfeld, Pradeep Ravikumar, and Andrej Risteski. The Risks of Invariant Risk Minimization. arXiv:2010.05761, 2020.

[3] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-Adversarial Training of Neural Networks. arXiv:1505.07818, 2015.

[4] Puwei Dai. Explanation of DANN and GRL. https://zhuanlan.zhihu.com/p/109051269, 2021.

[5] Yearn. Invariant Risk Minimization Reading Notes. https://zhuanlan.zhihu.com/p/273209891, 2022.

[6] Martin Arjovsky et al. Code Repository for Invariant Risk Minimization. https://github.com/facebookresearch/InvariantRiskMinimization, 2020.