주제 : Semi-Supervised Learning
(논문 : Learning by Association A versatile semi-supervised training method for neural networks)

이번에 진행할 Semi-Supervised Learning 주제이다.

Association이라고 하면 심리학이나 인지 공학 분야에서 주로 사용되는 용어이며, 해당 논문 또한 Association에서 착안하여 학습을 진행한다는 내용이다.

Association이라고 하면 우리는 관계라는 생각을 하게 되는데, 이는 Semi-Supervised Learning 관점에서 연관성을 통해서 학습이 가능하다는 개념에는 잘 부합하는 내용이다. 즉, 여려 종류의 자동차 사진의 예시를 보여주면 추후에는 레이블이 없는 자동차 사진을 보고 자동차라고 유추할 수 있다는 개념이다.

최근 방대한 데이터에는 레이블이 없는 경우가 많으며, 레이블을 하기 위해서는 많은 비용이 추가로 발생할 수 있다. 그렇기에 Semi-Supervised Learning 에서는 해당 비용을 낮출 수 있는 알고리즘에 대해 공부해보고 코드를 구현해 보도록 하겠다.

1. Overview

Association 개념을 Semi-Supervised Learning에 적용하는 방법을 논하기에 앞서 해당 논문에서 설정한 가정 한가지를 살펴보겠다.

> "네트워크가 Embedding Feature Vector를 잘 구성하였다면,
> 동일 Class의 경우 Feature Space에서 Vector간 서로 유사할 것이다."

당연하다고 생각 할 수도 있는 해당 개념에서부터 문제를 푸는 전략이 생긴다. Loss Function을 잘 만들어서 Labeled와 Unlabeled Data가 embedding feature space에서 서로 유사한 data들은 가깝고 다른 특성의 data들 간의 거리가 멀도록 네트워크를 구축하는 것이 목표이다.

그러면 자연스럽게 embedding space에서 각각의 data 사이에 가깝다 멀다를 어떻게 정의해야 할지에 대한 의문이 생긴다. 이 논문에서는 이를 다음과 같이 정의했다.

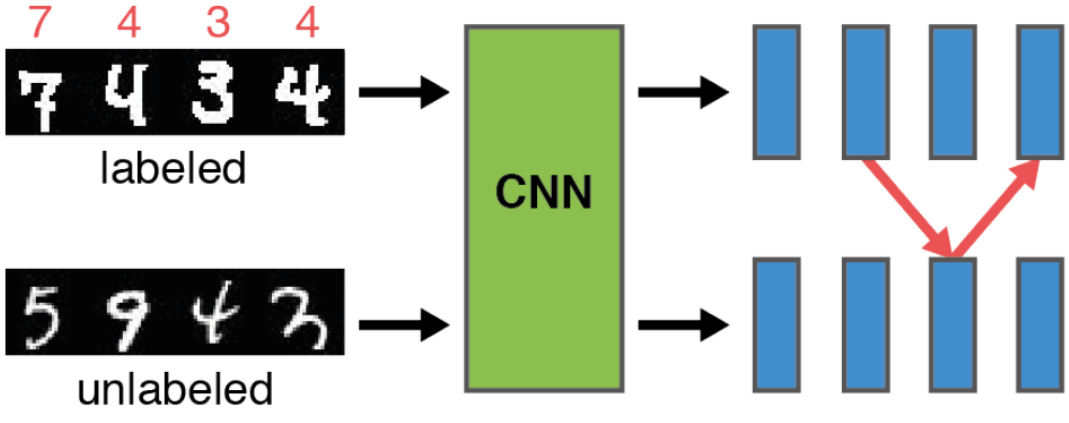
A : Labeled Data / B : Unlabeled Data의 Feature Space Vector를 나타낼 때,

A_i와 B_j사이의 유사도 M_{ij}는 다음과 같이 내적(inner product)로 정의한다.

$$M_{ij} = A_i \cdot B_j$$

여기까지만 보면 기존 Semi-Supervised Learning과 전혀 다를 것이 없지만, 앞으로 소개할 "Association"이란 개념을 포함하여 학습시키는 부분을 확인하면 차이가 있을 것이다.

여기서부터 중요하다. 데이터가 들어왔을 때, embedding을 해주는 네트워크(CNN)가 있으면, 이로부터 feature space가 만들어진다. 이런 embedding space에서 labeled data와 unlabeled data 사이의 "연관성 " 을 정량화 하기 위해 이 논문에서는 "walking"이라는 방법을 사용한다.



위 그림에서도 알 수 있듯이 labeled data의 feature vector에서 unlabeled data의 feature vector로 갔다(visit)가 다시 labeled data의 feature vector로 돌아왔을 때(walk) labeled data의 class가 바뀌지 않도록 제약을 주는 방식으로 loss 함수를 디자인한다. 이 얘기들을 좀더 수식화 하여 나타내면 아래와 같다.

- Transition Probability

$$P_{ij}^{ab} = P\left(B_j \middle| A_i\right) \coloneqq \left(softmax_{cols}(M)\right)_{ij}$$
$$= \exp\left(M_{ij}\right) / \sum_{j'} \exp(M_{ij'})$$

- Round Trip Probability

$$P_{ij}^{aba} \coloneqq \left(P^{ab}P^{ba}\right)_{ij}$$
$$= \sum_{k} P_{ik}^{ab} P_{kj}^{ba}$$

- Probability for correct walks

$$P(correct\ walk) = \frac{1}{|A|} \sum_{i \sim j} P_{ij}^{aba}$$
$$where\ i \sim j \iff class(A_i) = class(A_j)$$

만약 우리가 가정대로 네트워크가 잘 학습을 하고 있다면 embedding feature space를 만들 때, walking하고 돌아왔을 때 여전히 class 가 유지되어야 한다는 제약 때문에 자연스럽게 목표를 달성할 수 있게 된다는 것이 핵심 idea이다.

이렇게 기본적으로 loss를 잘 디자인해서 "연관성"이라는 추상적인 개념을 실제 계산이 가능한 형태로 잘 녹여낸 것이 이 논문의 독특한 점이라고 할 수 있다.

$$\mathcal{L}_{total} = \mathcal{L}_{walker} + \mathcal{L}_{visit} + \mathcal{L}_{classification}$$

총 세 부분으로 loss 함수가 나누어져 있는 것을 알 수 있는데, 사실 이를 $\mathcal{L}_{walker} + \mathcal{L}_{visit}$와 $\mathcal{L}_{classification}$ 이렇게 두 부분으로 구분하여 보는 것이 이해하기 편하다.

부분

- $\mathcal{L}_{walker} + \mathcal{L}_{visit}$
  - "Association(연관성)"을 표현하는 loss 함수 부분
- $\mathcal{L}_{classification}$
  - labeled data에 대한 사용되는 classification loss 함수 부분

코드 구현은 아래와 같다.

walker weight와 visit weight를 가지고 전체 Loss 함수를 구현한 부분이다. 또한 a-b, b-a, a-b-a에 대한 loss를 구현하고 전체 loss 함수는 cross entropy로 구현되었다.

```
def add_semisup_loss(self, a, b, labels, walker_weight=1.0, visit_weight=1.0):

    equality_matrix = tf.equal(tf.reshape(labels, [-1, 1]), labels)
    equality_matrix = tf.cast(equality_matrix, tf.float32)
    p_target = (equality_matrix / tf.reduce_sum(
        equality_matrix, [1], keep_dims=True))

    match_ab = tf.matmul(a, b, transpose_b=True, name='match_ab')
    p_ab = tf.nn.softmax(match_ab, name='p_ab')
    p_ba = tf.nn.softmax(tf.transpose(match_ab), name='p_ba')
    p_aba = tf.matmul(p_ab, p_ba, name='p_aba')

    self.create_walk_statistics(p_aba, equality_matrix)

    loss_aba = tf.losses.softmax_cross_entropy(
        p_target,
        tf.log(1e-8 + p_aba),
        weights=walker_weight,
        scope='loss_aba')
    self.add_visit_loss(p_ab, visit_weight)

    tf.summary.scalar('Loss_aba', loss_aba)
```

2. Walker loss 개념 & 코드 구현

여기서 walk라는 이름은 graph theory 부분에서 사용되는 용어로 graph theory 영역의 공부를 하다 보면 data 하나를 점으로 보았을 때, 한 점에서 다른 점으로 가는 것을 "walk"라는 용어로 표현한다.

위 그림이 walker loss에 대해 잘 설명해주는 부분이다. labeled data의 class인 "자동차"에서 unlabeled loss를 방문한 후 다시 labeled data로 돌아갔을 때 class가 여전히 "자동차"로 유지되길 바라는 부분이다. 여기서 주의할 점은 돌아온 labeled data가 꼭 시작점의 labeled data와 정확히 일치할 필요는 없지만, class는 유지되기를 바라면서 진행된다.

따라서 $\mathcal{L}_{walker}$에서는 만약 갔다가 돌아온 class가 달라지면 penalty를 주게 디자인 되어 있다.

$$\mathcal{L}_{walker} := H(T, P^{aba})$$

여기서 H는 cross entropy이고 T_{ij}는 class(A_i) =class(A_j)일 때, 1/#class(A_i)이고 아닐 때는 0인 uniform distribution이다. P^{aba}가 닮기를 바라는 T가 uniform distribution인 이유는 동일한 class로만 돌아오면 언제나 값이 같도록 하고 싶기 때문이다.(동일한 class의 다른 이미지로 돌아왔다고 penalty를 주고 싶지 않음)

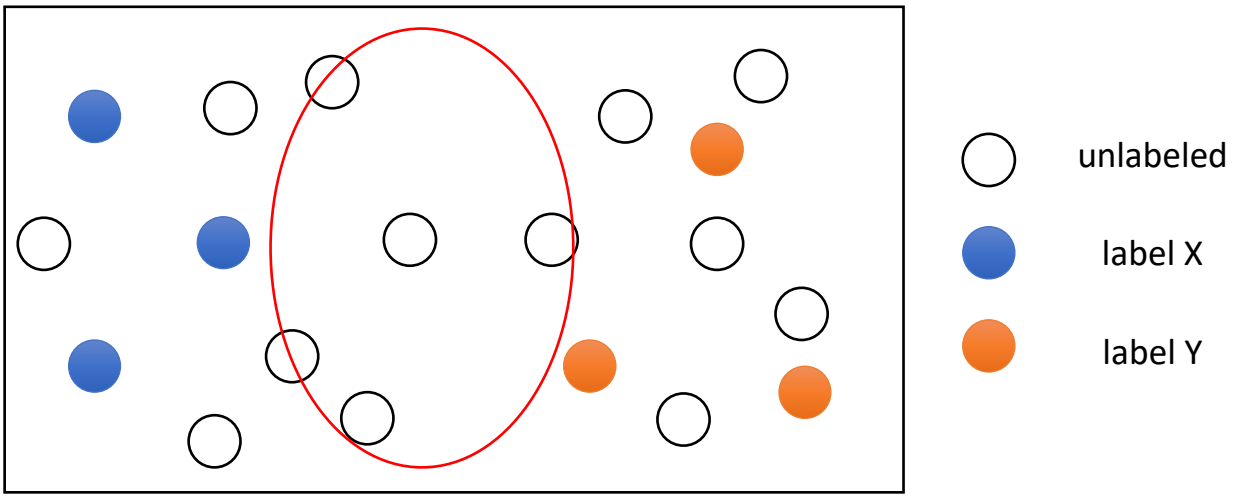
```
def create_walk_statistics(self, p_aba, equality_matrix):

  per_row_accuracy = 1.0 - tf.reduce_sum((equality_matrix * p_aba), 1)**0.5
  estimate_error = tf.reduce_mean(
      1.0 - per_row_accuracy, name=p_aba.name[:-2] + '_esterr')
  self.add_average(estimate_error)
  self.add_average(p_aba)

  tf.summary.scalar('Stats_EstError', estimate_error)
```

3. Visit loss 개념 & 코드 구현

$$\mathcal{L}_{visit} := \left(V, P^{visit}\right)$$

$$where\ P_j{}^{visit} := < P_{ij}{}^{aba} >_i, \qquad V_j := \frac{1}{|B|}$$

이 부분은 하고자 하는 것도 그리 어렵지 않다. 최대한 많은 sample을 다 골고루 봤으면 좋겠다는 것이다. 대부분의 semi-supervised learning 방식에서는 자기가 고른 labeled data를 기준으로 가까이 있는 unlabeled data만 보는데 그러지 말고 모두 다 보자는 것이다. 그러한 이유로 V가 uniform distribution이다. 그림을 통해 알아보겠다.



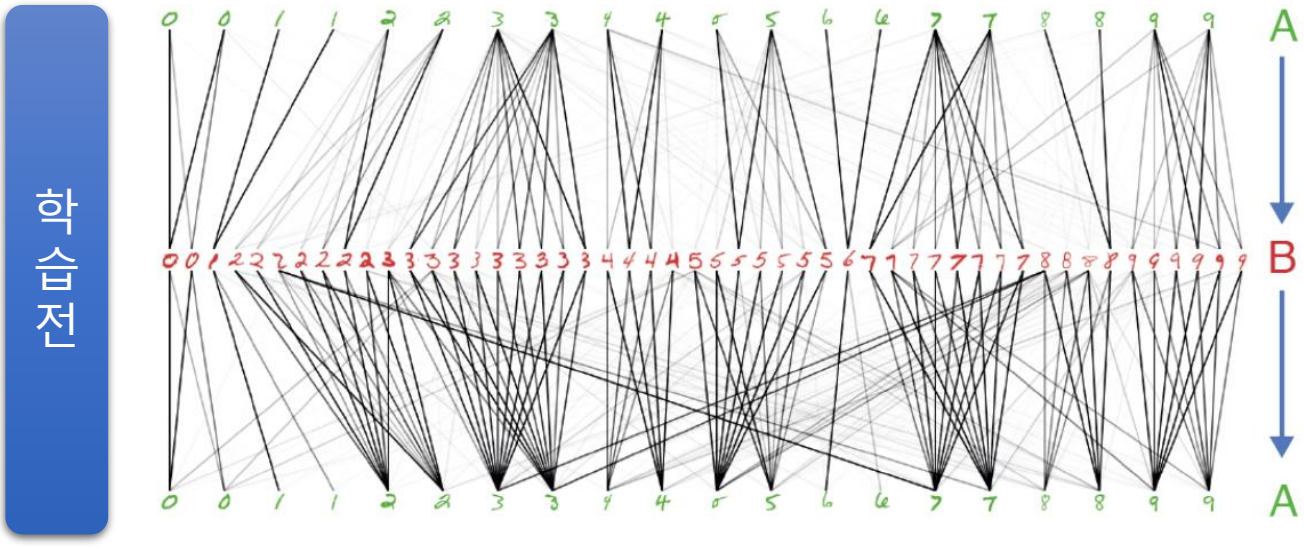
그림의 중간 동그라미 부분에 들어가는 data처럼 애매한 부분도 효과적으로 활용하고 싶다는 것이다. 단, 여기서 unlabeled data 너무 다른 경우 이 visit loss가 악영향을 끼치기 때문에 적절히 weight를 주어야할 필요가 있다.

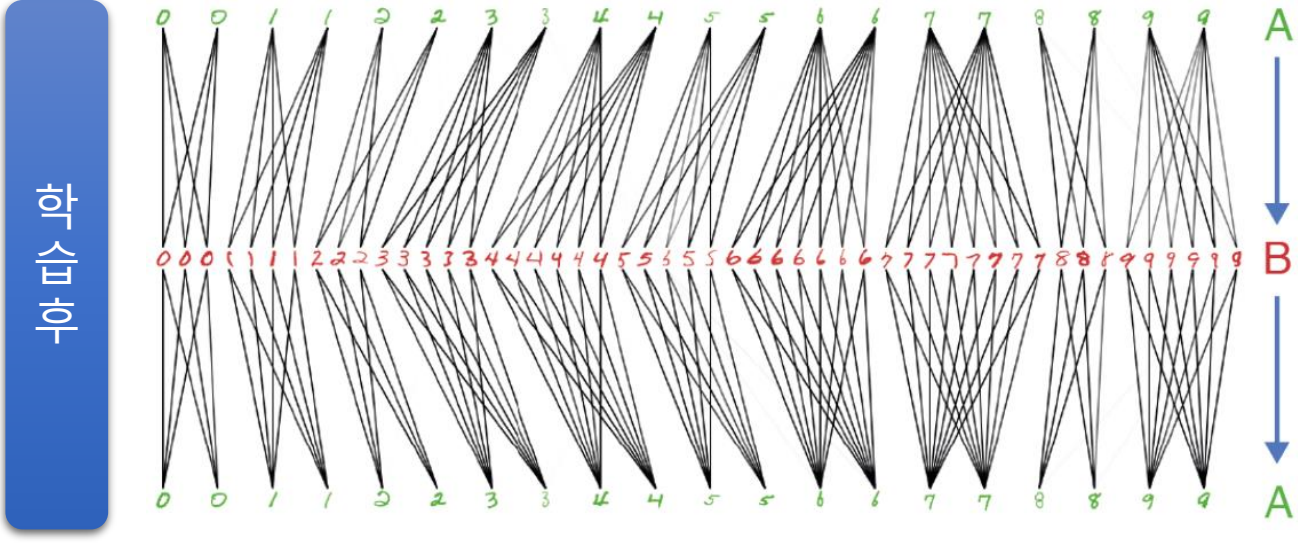
```
def add_visit_loss(self, p, weight=1.0):

  visit_probability = tf.reduce_mean(
      p, [0], keep_dims=True, name='visit_prob')
  t_nb = tf.shape(p)[1]
  visit_loss = tf.losses.softmax_cross_entropy(
      tf.fill([1, t_nb], 1.0 / tf.cast(t_nb, tf.float32)),
      tf.log(1e-8 + visit_probability),
      weights=weight,
      scope='loss_visit')

  tf.summary.scalar('Loss_Visit', visit_loss)
```

4. 실험 결과

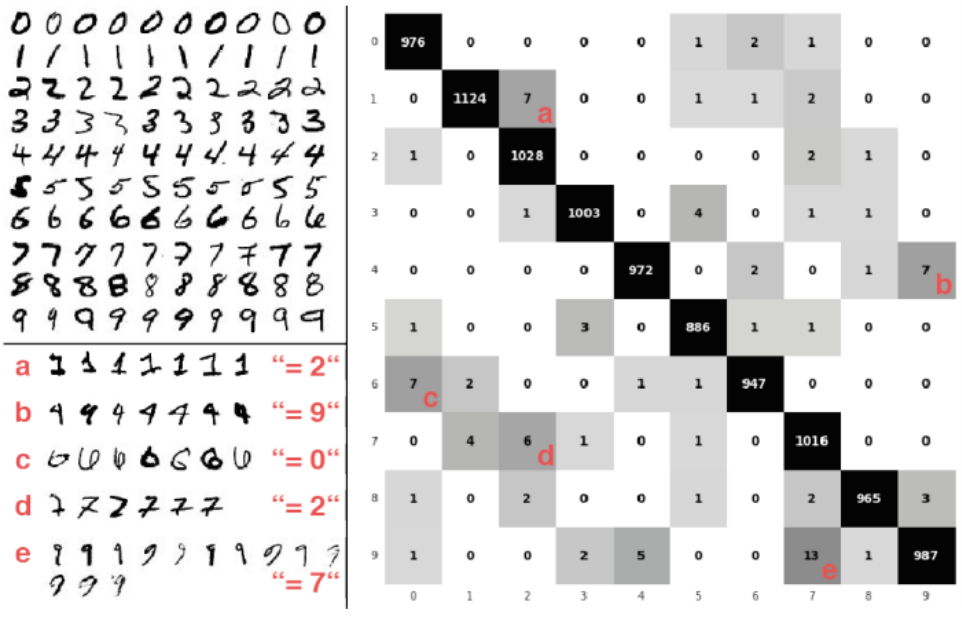
[MNIST]

먼저 MNIST 결과에 대해 학습 전후로 “Association”이 어떻게 바뀌어 가는지 시각화해서 보면 다음과 같다.





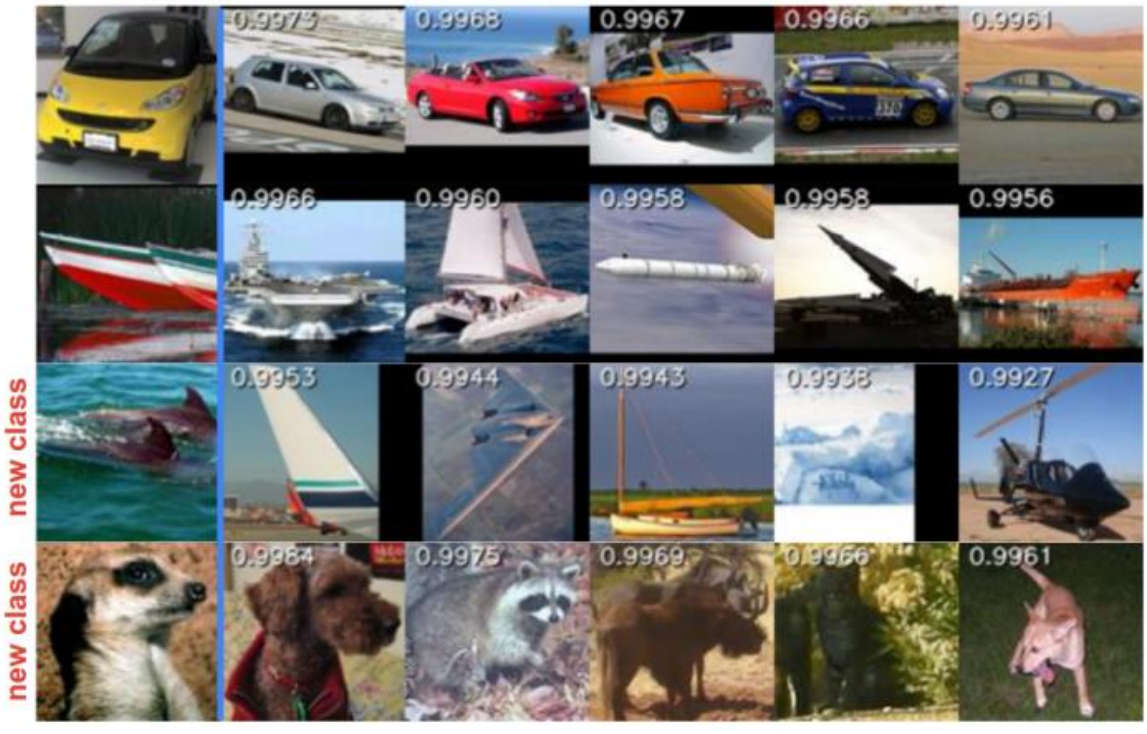
"학습전"은 학습을 시작해서 아주 약간 iteration을 돌렸을 때고, "학습후"가 네트워크가 수렴한 후를 의미한다.



이 실험 이후 MNIST에서 분류가 얼마나 잘 되었는지 확인을 해보면 test error가 0.96%로 매우 낮게 확인되었다. 심지어 이렇게 틀린 경우도 설명이 가능하다고 한다. 우측에서 보이는 것이 confusion matrix인데 여기서 틀린 부분을 좌하단에서 가져와 보여주면 사람이 봐도 왜 틀렸는지 이해가 될만한 비슷비슷한 숫자들을 헷갈린 것이라고 하고 있다.

[STL-10]

STL-10은 RGB 이미지로 10개의 class가 있는 데이터 셋이다. 약 5000개의 labeled 이미지와 10만개의 unlabeled 이미지가 있다. 재미있는 점은 이 unlabeled 이미지에 labeled 이미지에 존재하지 않는 class의 이미지들도 존재한다.



결과는 위 두 줄이 labeled 이미지에 class가 있는 이미지의 nearest neighbor를 5개 뽑아본 것으로 상당히 잘 되는 것을 확인할 수 있다. 아래 두 줄이 new class에 대한 부분으로 labeled 이미지 데이터 셋에 존재하지 않는 class인 돌고래와 미어켓을 입력으로 넣으니 네트워크가 내놓은 5개의 nearest neighbor인데, 나름 비슷하다고 생각한다. 돌고래의 삼각 지느러미 부분이 돛이나 비행기의 날개와 비슷하다 생각했는지 그런 이미지들이 같이 있고 미어켓은 신기하게도 동물들을 뽑아 온것을 확인할 수 있다.

[SVHN]

SVHN 데이터 셋에 대해 적용한 결과는 아래와 같다.

# labeled samples	# unlabeled samples			
	0	1000	20000	all
20	81.00 (3.01)	81.98 (2.58)	82.15 (1.35)	82.10 (1.91)
100	55.64 (6.54)	39.85 (7.19)	24.31 (7.19)	23.18 (7.41)
500	17.75 (0.65)	12.78 (0.99)	6.61 (0.32)	6.25 (0.32)
1000	10.92 (0.24)	9.10 (0.37)	5.48 (0.34)	5.14 (0.17)
2000	8.25 (0.32)	7.27 (0.43)	4.83 (0.15)	4.60 (0.21)
all	3.09 (0.06)	2.79 (0.02)	2.80 (0.03)	2.69 (0.05)

최소 labeled 샘플 갯수

Supervised 결과

이 테이블이 보여주는 점은 자신들의 method가 unlabeled 데이터셋이 점점 많이 주어질 수록 에러율이 매우 내려간다는 것이다. 즉, unlabeled data로부터 실제로 연관 정보를 잘 뽑아내고 있다는 것이다.

더욱 놀라운 점은 SVHN data로 MNIST data에 대한 Domain Adaptation 효과를 보여준 것이다. 아래 테이블을 보면 각각 SVHN에서만 학습시켰을 때, SVHN에서 학습시킨 후 MNIST로 Domain Adaptation 시켜줬을 때, MNIST에서만 학습시켰을 때, 세 가지 경우에서 MNIST 데이터셋에 대한 classification error를 알 수 있다.

	Data	Method	Domain (source → target) SVHN → MNIST
지도 학습	Source only	DA [8]	45.10
		DS [2]	40.8
		Ours	18.56
Domain Adaptation	Adapted	DA [8]	26.15 (42.6%)
		DS [2]	17.3 (58.3%)
		Ours	**0.51 (99.3%)**
지도 학습	Target only	DA [8]	0.58
		DS [2]	0.5
		Ours	0.38

이를 자신들의 method와 Domain Adaptation에서 유명했던 DANN, Domain Separation Nets 두 가지와 성능을 비교했는데 상당히 잘 되는 것을 볼 수 있다.

5. 코드 구현

MNIST 실험에 대한 코드는 아래와 같다.

```
def main(_):
  train_images, train_labels = mnist_tools.get_data('train')
  test_images, test_labels = mnist_tools.get_data('test')

  # Sample labeled training subset.
  seed = FLAGS.sup_seed if FLAGS.sup_seed != -1 else None
  sup_by_label = sample_by_label(train_images, train_labels,
                                 FLAGS.sup_per_class, NUM_LABELS, seed)

  graph = tf.Graph()
  with graph.as_default():
    model = SemisupModel(architectures.mnist_model, NUM_LABELS,
                         IMAGE_SHAPE)

    # Set up inputs.
    t_unsup_images, _ = create_input(train_images, train_labels,
                                     FLAGS.unsup_batch_size)
    t_sup_images, t_sup_labels = create_per_class_inputs(
        sup_by_label, FLAGS.sup_per_batch)

    # Compute embeddings and logits.
    t_sup_emb = model.image_to_embedding(t_sup_images)
    t_unsup_emb = model.image_to_embedding(t_unsup_images)
    t_sup_logit = model.embedding_to_logit(t_sup_emb)
```

```
    # Add losses.
    model.add_semisup_loss(
        t_sup_emb, t_unsup_emb, t_sup_labels, visit_weight=FLAGS.visit_weight)
    model.add_logit_loss(t_sup_logit, t_sup_labels)

    t_learning_rate = tf.train.exponential_decay(
        FLAGS.learning_rate,
        model.step,
        FLAGS.decay_steps,
        FLAGS.decay_factor,
        staircase=True)
    train_op = model.create_train_op(t_learning_rate)
    summary_op = tf.summary.merge_all()

    summary_writer = tf.summary.FileWriter(FLAGS.logdir, graph)

    saver = tf.train.Saver()
  with tf.Session(graph=graph) as sess:
    tf.global_variables_initializer().run()

    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)

    for step in range(FLAGS.max_steps):
      _, summaries = sess.run([train_op, summary_op])
      if (step + 1) % FLAGS.eval_interval == 0 or step == 99:
        print('Step: %d' % step)
        test_pred = model.classify(test_images).argmax(-1)
        conf_mtx = confusion_matrix(test_labels, test_pred, NUM_LABELS)
        test_err = (test_labels != test_pred).mean() * 100
        print(conf_mtx)
        print('Test error: %.2f %%' % test_err)
        print()

        test_summary = tf.Summary(
            value=[tf.Summary.Value(
                tag='Test Err', simple_value=test_err)])

        summary_writer.add_summary(summaries, step)
        summary_writer.add_summary(test_summary, step)

        saver.save(sess, FLAGS.logdir, model.step)

    coord.request_stop()
    coord.join(threads)


if __name__ == '__main__':
  app.run()
```

6. 마무리

① 단순하지만 신선한 Semi-Supervised training method 제안

② Tensorflow Impletation이 있고 arbitrary network architecture에 add-on처럼 범용적으로 붙여 사용 가능

③ 당시 SOTA method에 비해 최대 64% 성능 향상

MNIST, STL-10, SVHN 결과에서는 논문 만큼은 아니었지만, 그래도 유사한 성능을 확인할 수 있었다. 하지만 “네트워크가 Embedding Feature Vector를 잘 구성하였다면, 동일 Class의 경우 Feature Space에서 Vector간 서로 유사할 것이다.”라는 전제가 깨지는 데이터 셋에 대한 추가 실험 결과도 추후 확인해 볼 필요가 있겠다.