



Published in Towards Data Science

You have 2 free member-only stories left this month. [Sign up for Medium and get an extra one](#)

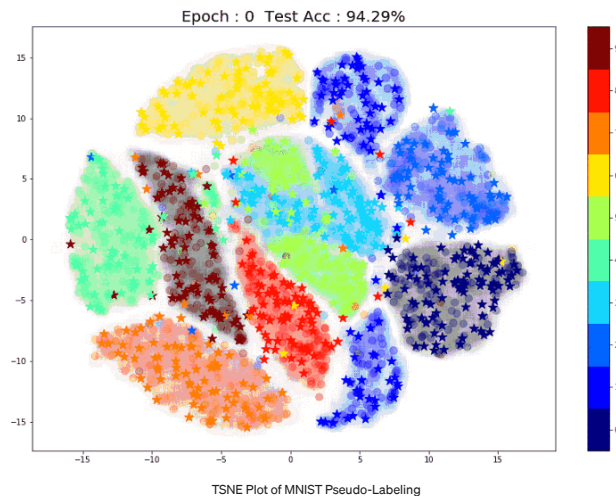
Anirudh Shenoy

Dec 3, 2019 · 16 min read · Listen



Pseudo-Labeling to deal with small datasets — What, Why & How?

A guide to using your model's output to improve your model's output!



A few days ago I came across Yoshua Bengio's reply to a Quora question — "[Why is Unsupervised Learning important?](#)". Here's an excerpt of his reply:

To climb the AI ladder with supervised learning may require "teaching" the computer all the concepts that matter to us by showing tons of examples where these concepts occur. This is not how humans learn: yes, thanks to language we get some examples illustrating new named concepts that are given to us, but the bulk of what we observe does not come labeled, at least initially.

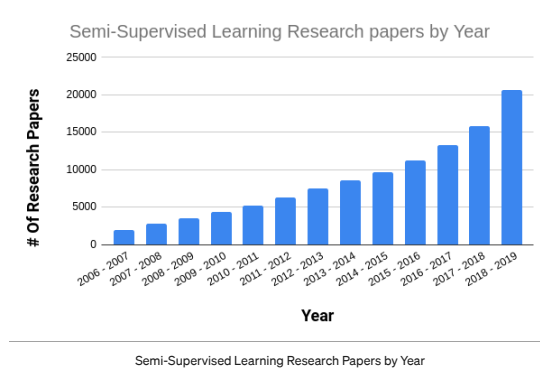
His reply makes a lot of sense both from a neuroscientific perspective and a practical perspective. Labeling data is expensive in terms of time as well as money. The obvious solution to this problem is to figure out a way to either:

- (a) Make ML algorithms work without labeled data (i.e. Unsupervised Learning)
- (b) Automatically label data or use large amounts of unlabeled data along with small amounts of labeled data (i.e. Semi-Supervised Learning)

Unsupervised learning is quite a difficult problem to solve, as Yann LeCun mentions in this [article](#):

"We know the ultimate answer is unsupervised learning, but we don't have the answer yet."

However, of late there has been renewed interest in Semi-Supervised Learning which is reflected in both academic and industrial research. Here's a graph showing the number of research papers related to Semi-Supervised Learning on Google Scholar by year.



In this blog, we'll take an in-depth look at Pseudo-Labeling — a simple Semi-Supervised Learning (SSL) algorithm. Although Pseudo-Labeling is a naive approach, it gives us an excellent opportunity to understand the challenges with SSL and provides a foundation to learn some of the modern improvements like MixMatch, Virtual Adversarial Training, etc.

Outline:

1. What is Pseudo-Labeling?
2. Understanding the Pseudo-Labeling method
3. Implementing Pseudo-Labeling
4. Why does Pseudo-Labeling work?
5. When does Pseudo-Labeling not work well?
6. Pseudo-Labeling with conventional ML algorithms
7. Challenges with Semi-Supervised Learning

1. What is Pseudo-Labeling?

First proposed by [Lee in 2013 \[1\]](#), the pseudo-labeling method uses a small set of labeled data along with a large amount of unlabeled data to improve a model's performance. The technique itself is incredibly simple and follows just 4 basic steps:

1. Train model on a batch of labeled data
2. Use the trained model to predict labels on a batch of unlabeled data
3. Use the predicted labels to calculate the loss on unlabeled data
4. Combine labeled loss with unlabeled loss and backpropagate

...and repeat.

This technique might seem quite strange — almost similar to the hundreds of “[free energy device](#)” [videos on youtube](#). However, Pseudo-Labeling has been successfully used on several problems. In fact, [in a Kaggle competition](#), a team used Pseudo-Labeling to improve their model's performance just enough to secure the 1st place and win \$25,000.

We'll take a look at why this works in a bit, for now, let's look at some details.

2. Understanding the Pseudo-Labeling method

Pseudo-labeling trains the network with labeled and unlabeled data simultaneously in each batch. This means for each batch of labeled and unlabeled data, the training loop does:

1. One single forward pass on the labeled batch to calculate the loss → This is the labeled loss
2. One forward pass on the unlabeled batch to predict the “pseudo labels” for the unlabeled batch
3. Use this “pseudo label” to calculate the unlabeled loss.

Now instead of simply adding the unlabeled loss with the labeled loss, Lee proposes using weights. The overall loss function looks like this:

$$L = \frac{1}{n} \sum_{m=1}^n \sum_{i=1}^C L(y_i^m, f_i^m) + \alpha(t) \frac{1}{n'} \sum_{m=1}^{n'} \sum_{i=1}^C L(y_i'^m, f_i'^m), \quad (15)$$

Equation [15] Lee (2013) [1]

Or in simpler words:

Loss per Batch = Labeled Loss + *Weight* * Unlabeled Loss

In the equation, the weight (alpha) is used to control the contribution of unlabeled data to the overall loss. In addition, the weight is a function of time (epochs) and is slowly increased during training. This allows the model to focus more on the labeled data initially when the performance of the classifier can be bad. As the model's performance increases over time (epochs), the weight increases and the unlabeled loss has more emphasis on the overall loss.

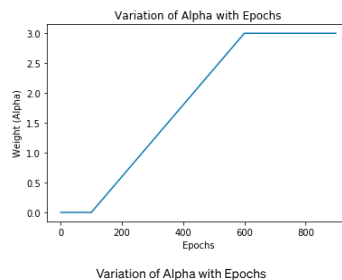
Lee proposes using the following equation for alpha (t) :

$$\alpha(t) = \begin{cases} 0 & t < T_1 \\ \frac{t-T_1}{T_2-T_1} \alpha_f & T_1 \leq t < T_2 \\ \alpha_f & T_2 \leq t \end{cases} \quad (16)$$

Equation [16] Lee (2013) [1]

where $\alpha_f = 3$, $T_1 = 100$ and $T_2 = 600$. All of these are hyperparameters that change based on the model and the dataset.

Let's check how Alpha changes with the epochs:



In the first T_1 epochs (100 in this case) the weight is 0 — effectively forcing the model to train only on the labeled data. After T_1 epochs, the weight linearly increases to α_f (3 in this case) until T_2 epochs (600 in this case) — this allows the model to slowly incorporate the unlabeled data. T_2 and α_f control the rate at which the weight increases and the value after saturation respectively.

If you're familiar with optimization theory you might recognize this equation from [Simulated Annealing](#).

And that's all there is to understand Pseudo-Labeling from an implementation perspective. The paper uses MNIST to report performance so we'll stick to the same dataset which will help us check if our implementation is working correctly.

3. Implementing Pseudo-Labeling

We'll use PyTorch 1.3 with CUDA for the implementation, although you should have no problems using Tensorflow/Keras as well.

Model Architecture:

While the paper uses a simple 3 fully-connected layer network, during testing I found that Conv Nets perform much better. We'll use a simple 2 Conv Layer + 2 Fully Connected Layer network with dropout (as described in this [repo](#))

```
1 import torch
2 from torch import nn
3 import torch.nn.functional as F
```

```
4
5 class Net(nn.Module):
6     def __init__(self):
7         super(Net, self).__init__()
8         self.conv1 = nn.Conv2d(1, 20, kernel_size=5)
9         self.conv2 = nn.Conv2d(20, 40, kernel_size=5)
10        self.conv2_drop = nn.Dropout2d()
11        self.fc1 = nn.Linear(640, 150)
12        self.fc2 = nn.Linear(150, 10)
13        self.log_softmax = nn.LogSoftmax(dim = 1)
14
15    def forward(self, x):
16        x = x.view(-1,1,28,28)
17        x = F.relu(F.max_pool2d(self.conv1(x), 2))
18        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
19        x = x.view(-1, 640)
20        x = F.relu(self.fc1(x))
21        x = F.dropout(x, training=self.training)
22        x = F.relu(self.fc2(x))
23        x = self.log_softmax(x)
24        return x
25
26 net = Net().cuda()
```

ConvNet_for_mnist.py hosted with ❤ by GitHub [view raw](#)

Model Architecture for MNIST

Baseline Performance:

We'll use 1000 labeled images(class balanced) and 59,000 unlabeled images for the train set and 10,000 images for the test set.

First, let's check the performance on the 1000 labeled images without using any of the unlabeled images (i.e. simple supervised training)



With 1000 labeled images the best test accuracy is 95.57%. Now that we have a baseline, let's go ahead with the pseudo-labeling implementation.

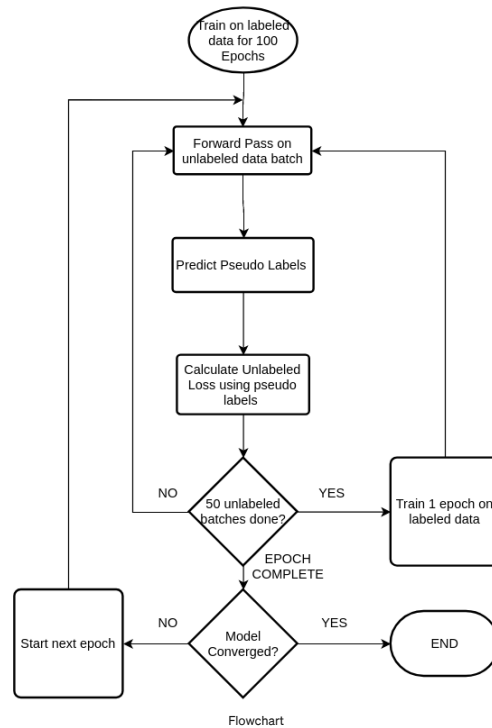
Pseudo-Labeling Implementation:

For the implementation we'll make 2 minor changes which make the code simpler and perform better:

- 1. In the first 100 epochs, we'll train the model on the labeled data as usual (no unlabeled data). As we've seen earlier, this makes no difference to pseudo-labeling since $\alpha = 0$ during this period anyway.
- 2. In the next 100+ epochs, we will train on the unlabeled data (with alpha weights). Here for every 50 unlabeled batches, we will train one epoch on the labeled data — this acts as a correcting factor.

Don't worry if this sounds confusing, it's much easier in code. This modification is based on this Github [repo](#) and it helps in 2 ways:

- 1. It reduces overfitting on the labeled training data
- 2. Improves speed since we need to make only 1 forward pass per batch (on the unlabeled data) instead of 2 (unlabeled and labeled) as mentioned in the paper.



```

1  # Based on https://github.com/peimengsui/semi_supervised_mnist
2  from tqdm import tqdm_notebook
3
4  T1 = 100
5  T2 = 700
6  af = 3
7
8  def alpha_weight(step):
9      if step < T1:
10         return 0.0
11     elif step > T2:
12         return af
13     else:
14         return ((step-T1) / (T2-T1))*af
15
16
17  def semisup_train(model, train_loader, unlabeled_loader, test_loader):
18      optimizer = torch.optim.SGD(model.parameters(), lr = 0.1)
19      EPOCHS = 100
20
21      # Instead of using current epoch we use a "step" variable to calculate alpha_weight
22      # This helps the model converge faster
23      step = 100
24
25      model.train()
26      for epoch in tqdm_notebook(range(EPOCHS)):
27          for batch_idx, x_unlabeled in enumerate(unlabeled_loader):
28
29
30              # Forward Pass to get the pseudo labels
31              x_unlabeled = x_unlabeled.cuda()
32              model.eval()
33              output_unlabeled = model(x_unlabeled)
34              _, pseudo_labeled = torch.max(output_unlabeled, 1)
35              model.train()
36
37              # Now calculate the unlabeled loss using the pseudo label
38              output = model(x_unlabeled)
39              unlabeled_loss = alpha_weight(step) * F.nll_loss(output, pseudo_labeled)
40
41              # Backpropagate
42              optimizer.zero_grad()
43              unlabeled_loss.backward()
44              optimizer.step()
45
46
47              # For every 50 batches train one epoch on labeled data
48              if batch_idx % 50 == 0:
49
50                  # Normal training procedure
51                  for batch_idx, (X_batch, y_batch) in enumerate(train_loader):
52                      X_batch = X_batch.cuda()
53                      y_batch = y_batch.cuda()
54                      output = model(X_batch)
55                      labeled_loss = F.nll_loss(output, y_batch)
56
57                      optimizer.zero_grad()
58                      labeled_loss.backward()
59                      optimizer.step()
60
61                  # Now we increment step by 1
62                  step += 1

```

semi_supervised_training.py hosted with ❤ by GitHub

[view raw](#)

Pseudo-Labeling Loop for MNIST

Note: I'm not including the code for the supervised training (first 100 epochs) here as it's very straightforward. You can find all the code in my [repo here](#)

Here's the result after training 100 epochs on labeled data followed by 170 epochs of semi-supervised training:

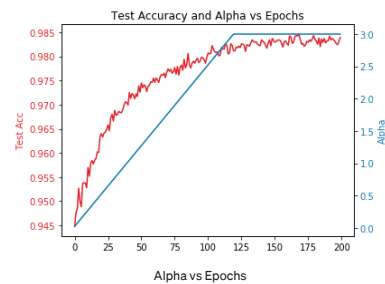
Best Accuracy is at 168 epochs

Epoch: 168 : Alpha Weight : 3.00000 | Test Acc : 98.46000 | Test Loss : 0.075

After using the unlabelled data we reached an accuracy of 98.46% that's ~ 3% more than with supervised training. In fact, our results are better than the results from the paper — 95.7% for 1000 labeled samples.

Let's do some visualization to understand how pseudo-labeling is working under the hood.

Alpha Weight vs Accuracy

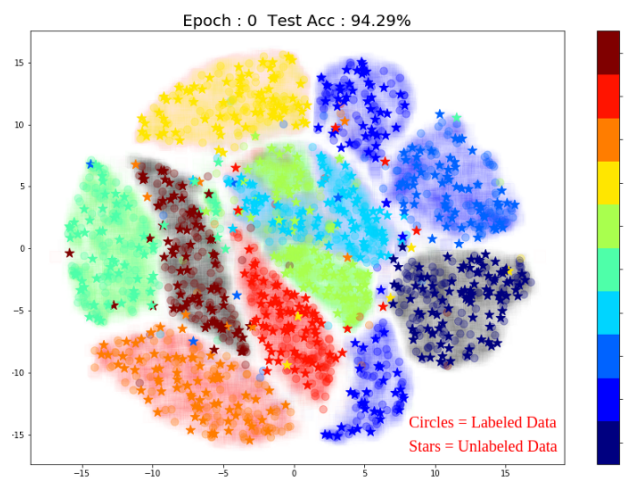


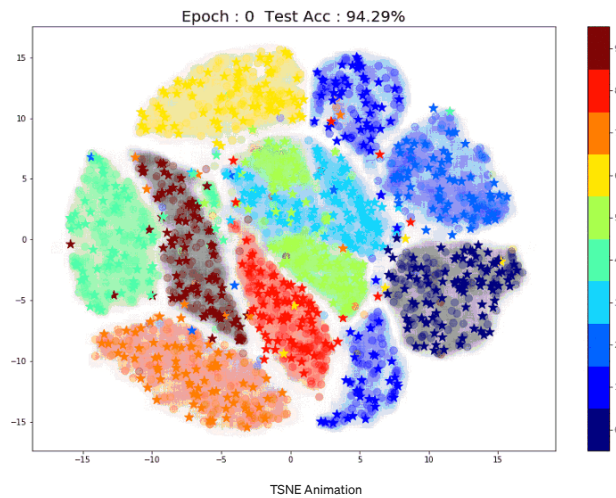
It's clear that as alpha increases the test accuracy also slowly increases and later saturates.

TSNE Visualization

Now let's have a look at how the pseudo labels are being assigned at every epoch. In the plot below, there are 3 things to note:

1. The faint color in the background of each cluster is the true label. This is created using TSNE of all 60k training images (labels used)
2. The small circles inside each cluster are from the 1000 training images that were used in the supervised training phase.
3. The small stars that keep moving are the pseudo labels that the model assigns for the unlabeled images for each epoch. (For each epoch I used ~750 randomly sampled unlabeled images to create the plot)

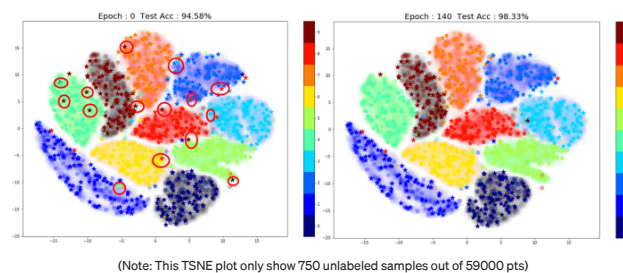




Here are some things to notice:

1. Most of the pseudo-labels are correct. (Stars are in clusters with the same color) This can be attributed to the high initial test accuracy.
2. As training continues, the percentage of correct pseudo labels increases. This is reflected in the increased overall test accuracy of the model.

Here's a plot that shows the **same** 750 points at Epoch 0 (left) and Epoch 140 (right). I've marked the points that have improved in red circles.



Now let's find out why pseudo labeling actually works.

4. Why does Pseudo-Labeling work?

The goal of any Semi-Supervised Learning algorithm is to use both the unlabeled and labeled samples to learn the underlying structure of the data. Pseudo-Labeling is able to do this by making two important assumptions:

1. **Continuity Assumption (Smoothness):** *Points that are close to each other are more likely to share a label.* ([Wikipedia](#)) In other words, small changes in input **do not** cause large changes in output. This assumption allows pseudo labeling to conclude that small changes in images like rotation, shearing, etc do not change the label.
2. **Cluster Assumption:** *The data tend to form discrete clusters, and points in the same cluster are more likely to share a label.* This is a special case of the continuity assumption ([Wikipedia](#)) Another way to look at this is — the decision boundary between classes lies in the low-density region (*doing so helps in generalization — similar to maximum margin classifiers like SVM*).

This is why the initial labeled data is important — it helps the model learn the underlying cluster structure. When we assign a pseudo label in the code, we are using the cluster structure that the model has learned to infer labels for the unlabeled data. As the training progresses, the learned cluster structure is improved using the unlabeled data.

If the initial labeled data is too small in size or contains outliers, pseudo labeling will likely assign incorrect labels to the unlabeled points. The opposite also holds, i.e. pseudo labeling can benefit from a classifier that is already performing well with just the labeled data.

This should make more sense in the next section when we look at scenarios where pseudo-labeling fails.

Get started

Sign In

Q Search



Anirudh Shenoy
150 Followers

Product @ Yellow Messenger |
Breaking down Machine Learning
& Data Science topics into simple
concepts |
[linkedin.com/in/anirudhshenoy/](https://www.linkedin.com/in/anirudhshenoy/) |
Views are personal |

Follow

More from Medium

Thuw... in Towar...
**Let the Script Find
Out the ML Model
that Outperform...**

Tarek ... in MLea...
**Data Science in
the next 1000
years.**

Mat... in Toward...
**Top Encoders for
Data Scientists**

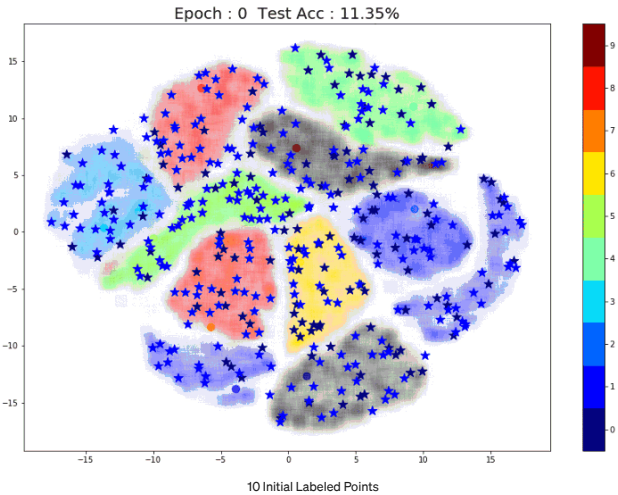
Ken... in Toward...
**Principal
Component
Regression—...**



5. When does Pseudo-Labeling not work well?

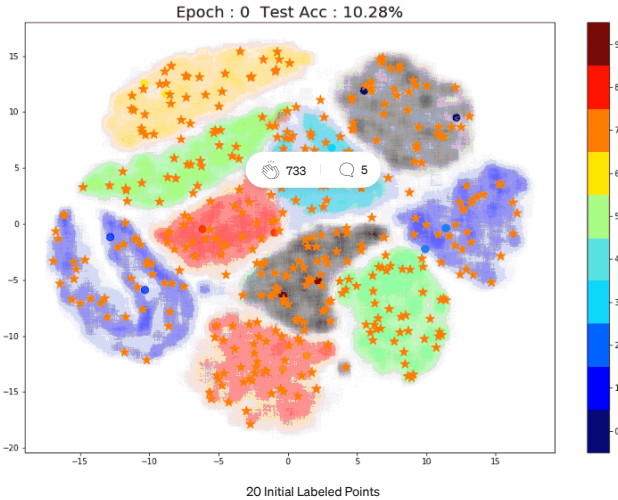
Initial Labeled data is not enough to determine clusters

To understand this scenario better, let's run a small experiment: Instead of using 1000 initial points let's take the extreme case and use just 10 labeled points and see how pseudo-labeling performs:

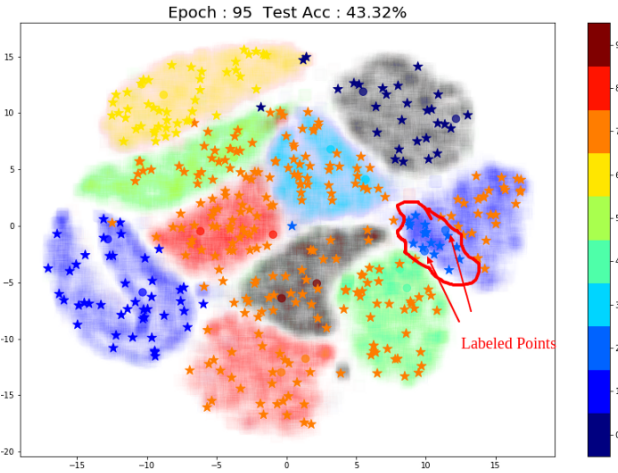


As expected, pseudo-labeling has almost no difference. The model itself is as good as a random model with 10% accuracy. Since each class effectively has just 1 point, the model is incapable of learning the underlying structure for any class.

Let's increase the number of labeled points to 20 (2 points per class) :

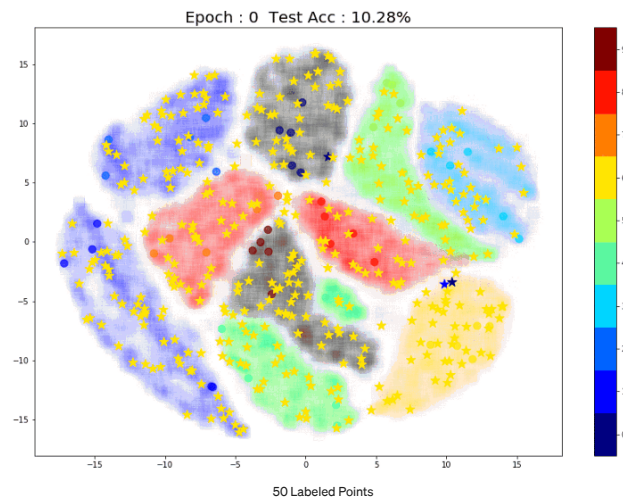


Now the model is performing slightly better as it learns the structure for some classes. Here's something interesting — notice that pseudo-labeling assigns the correct labels for these points (marked in red in the image below) most likely because there are two labeled points closeby.

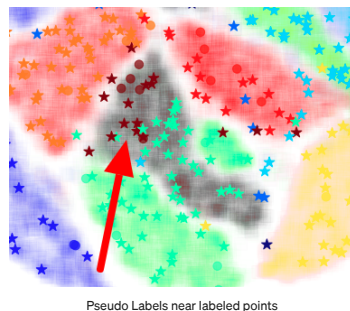


Small Cluster near labeled points

And finally, let's try 50 points:



The performance is much better! And once again, notice the small group of brown labeled points right in the center of the images. The points in the same brown cluster but further away from the labeled points are always incorrectly predicted as Aqua green ('4') or Orange ('7').



A few things to note:

1. For all the above experiments (10,20 and 50 points) the way the labeled points were chosen made a huge difference. Any outliers completely changed the model's performance and predictions for pseudo-labels. This is a common problem with small datasets. (You can read [my previous blog](#) where I've discussed this in detail)
2. While TSNE is a great tool for visualization, we need to keep in mind that it is *probabilistic* and merely gives us an *idea* of how clusters might be distributed in higher-dimensional space.

To conclude, both the quantity and quality of initial labeled points make a difference when it comes to pseudo-labeling. Further, the model might require different amounts of data for different classes to understand that particular class's structure.

Initial Labeled data does not include some classes

Let's see what happens if the labeled dataset does not contain one class (eg: '7' not included in the labeled set, but the unlabeled data still retains all classes)

After training 100 epochs on the labeled data:

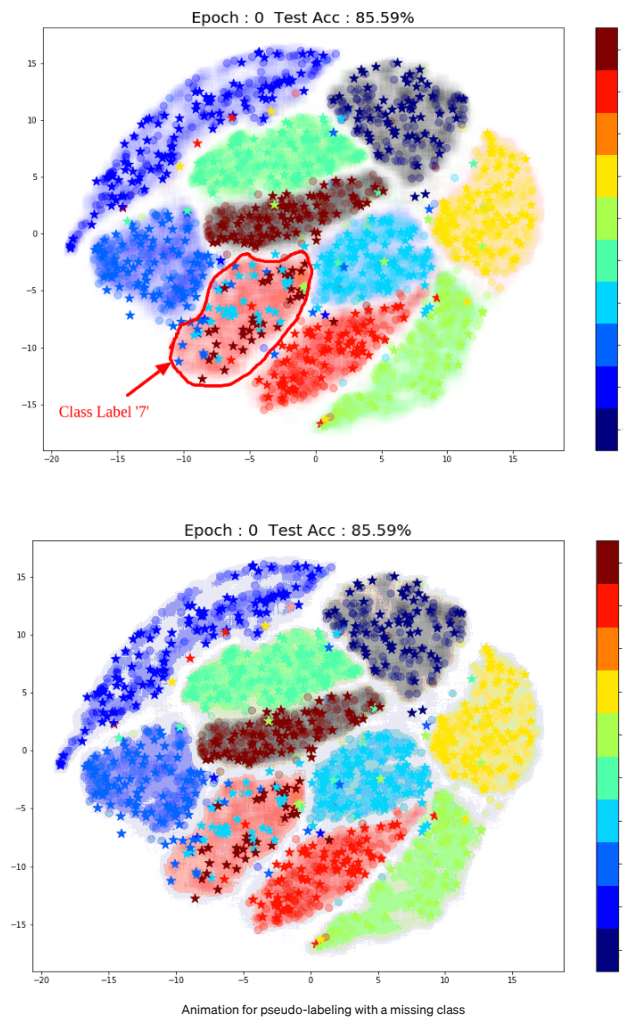
Test Acc : 85.63000 | Test Loss : 1.555

And after semi-supervised training :

Epoch: 99 : Alpha Weight : 2.50000 | Test Acc : 87.98000 | Test Loss : 2.987

The overall accuracy does increase from 85.6% to 87.98% but does not show any improvements after that. This is obviously because the model is unable to learn the cluster structure for the class label "7".

The animations below should make these clear:

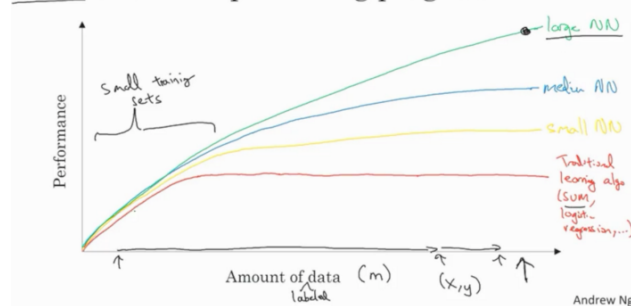


It's no surprise that pseudo-labeling struggles here as our model does not have the capability to learn about classes that it has never seen before. However, over the past few years, a lot of interest has been shown in Zero-Shot Learning techniques which enable models to recognize labels even if they do not exist in the training data.

No benefit from increased data

In some cases, the model might not have enough complexity to take advantage of the additional data. This usually happens when using pseudo-labeling with conventional ML algorithms like Logistic Regression or SVMs. When it comes to Deep Learning models, as Andrew Ng mentions in his Coursera course — Large DL models almost always benefit from having more data.

Scale drives deep learning progress



Andrew Ng — Coursera Deep Learning Specialization

6. Pseudo-Labeling with conventional ML algorithms

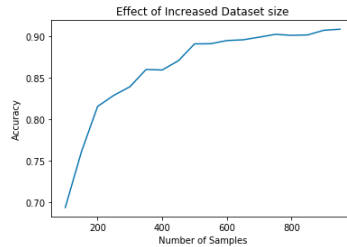
In this section, we'll apply the pseudo-labeling concept to Logistic Regression. We'll use the same MNIST dataset with 1000 labeled images, 59000 unlabeled images, and 10000 test images.

Feature Engineering

We'll first normalize all the images, followed by PCA decomposition from 784 dimensions to 50 dimensions. Following this, we'll use sklearn's `PolynomialFeatures()` with `degree = 2` to add interaction and quadratic features. This leaves us with 1326 features per data point.

Effect of Increased Dataset size

Before we get started on pseudo-labeling let's check how Logistic Regression performs when the training dataset size is slowly increased. This will help us understand if the model can benefit from pseudo-labeling.



As the number of samples in the training dataset increases from 100 to 1000 we see that the accuracy slowly increases. Further, it looks like the accuracy is not stagnating and is following an upward trend. From this, we can conclude that pseudo-labeling should give us a boost in performance here.

Baseline Performance:

Let's check the test accuracy when Logistic Regression uses only the 1000 labeled images. We'll do 10 training runs to account for any variations in test scores.

```
from sklearn.linear_model import SGDClassifier

test_acc = []
for _ in range(10):
    log_reg = SGDClassifier(loss='log', n_jobs=-1, alpha=1e-5)
    log_reg.fit(x_train_poly, y_train)
    y_test_pred = log_reg.predict(x_test_poly)
    test_acc.append(accuracy_score(y_test_pred, y_test))

print('Test Accuracy: {:.2f}%'.format(np.array(test_acc).mean()*100))
```

Output:
Test Accuracy: 90.86%

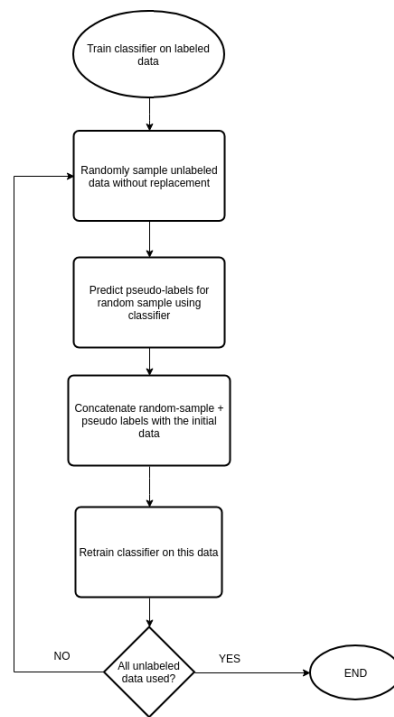
Our baseline test accuracy using Logistic Regression is 90.86%

Pseudo-Labeling Implementation:

When working with Logistic Regression and other conventional ML algorithms we need to use pseudo-labeling in a slightly different way, although the concept remains the same.

Here are the steps:

1. We first train a classifier on our labeled train set.
2. Next, we use this classifier to predict the labels on a randomly sampled set from the unlabeled dataset.
3. We combine both the original train set and the predicted set and retrain the classifier on this new dataset.
4. Repeat steps 2 and 3 until all of the unlabeled data has been used.



This technique is slightly similar to the one mentioned in this [blog](#). However, here we recursively generate pseudo labels until all unlabeled data have been used. Thresholds can also be used to ensure that pseudo labels are generated only for points that the model is very confident about (though this is not necessary).

Here's the implementation as a wrapper around a sklearn estimator:

Pseudo-Labeling Wrapper for Logistic Regression

(Complete code is available in the [repo](#))

And now we can use it with Logistic Regression :

```

from sklearn.linear_model import SGDClassifier

log_reg = SGDClassifier(loss = 'log', n_jobs = -1, alpha = 1e-5)

pseudo_labeller = pseudo_labeling(
    log_reg,
    x_unlabeled_poly,
    sample_rate = 0.04,
    verbose = True
)

pseudo_labeller.fit(x_train_poly, y_train)
  
```

```
y_test_pred = pseudo_labeller.predict(x_test_poly)
print('Test Accuracy: {:.2f}%'.format(accuracy_score(y_test_pred,
y_test)*100))
```

Output:
Test Accuracy: 92.42%

Pseudo-Labeling increased the accuracy from 90.86% to 92.42%. (*Non-linear models with higher complexity like XGBoost might perform better*)

Here, `sample_rate` is similar to `alpha(t)` from the deep learning model example. Previously, `alpha` was used to control the amount of unlabeled loss that was used, while in this case `sample_rate` controls how many unlabeled points are used in each iteration.

The `sample_rate` value itself is a hyperparameter that needs to be tuned based on the dataset and model (*similar to T1, T2, and alpha_f*). A value of 0.04 worked best for the MNIST + Logistic Regression example.

An interesting modification would be to schedule `sample_rate` to ramp up as the training progresses exactly similar to `alpha(t)`.

Before we conclude, let's look at some of the challenges in Semi-Supervised Learning in general.

7. Challenges with Semi-Supervised Learning

Combining Unlabeled Data with Labeled Data

The primary objective of Semi-Supervised Learning is to use the unlabeled data along with the labeled data to understand the underlying structure of the dataset. The obvious question here is — How to utilize the unlabeled data to achieve this purpose?

In the Pseudo-Labeling technique, we saw that a scheduled weight function (`alpha`) was used to slowly combine the unlabeled data with the labeled data. However, the `alpha(t)` function assumes that the model confidence increases over time and therefore increases the unlabeled loss linearly. This need not be the case as model predictions can sometimes be incorrect. In fact, if the model makes several wrong unlabeled predictions, pseudo-labeling can act like a bad feedback loop and deteriorate performance further. (*Ref: Section 3.1 Arazo et al 2019 [2]*)

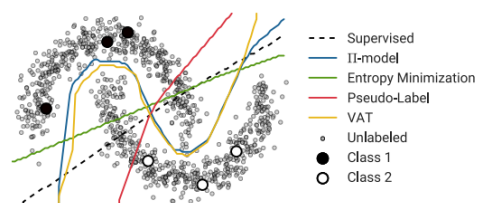
One solution for the problem above is to use probability thresholds — similar to what we did with Logistic Regression.

Other Semi-Supervised Learning algorithms use different ways to combine the data, for example, MixMatch uses a 2-step process for guessing the label (for the unlabeled data) followed by MixUp data augmentation to combine the unlabeled data with the labeled data. (*Berthelot et al (2019) [3]*)

Data Efficiency

Another challenge with Semi-Supervised Learning is to design algorithms that can work with very small amounts of labeled data. As we've seen with pseudo labeling, the model works best with 1000 initial labeled samples. However, when the labeled dataset is reduced further (for eg: with 50 points), pseudo-labeling's performance starts to drop.

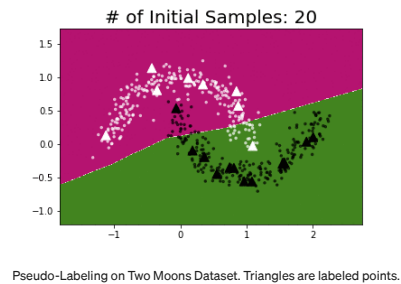
Oliver et al. (2018) [4] did a comparison of several Semi-Supervised Learning algorithms and found that Pseudo-Labeling fails on the “two-moons” dataset while other models like VAT and pi-model worked much better.



Source: Oliver et al (2018) [4]

As shown in the image, VAT and Pi-Model learn a decision boundary that is surprisingly good with just 6 labeled data points (shown in large white and black circles). Pseudo-Labeling on the other hand completely fails and learns a linear decision boundary instead.

I repeated the experiment using the same model that Oliver et al. used and found that pseudo-labeling required anywhere from 30–50 labeled points (depending on the position of the labeled points) to learn the underlying data structure.



To make Semi-Supervised Learning more practical we need algorithms that are highly data-efficient i.e. ones that can work on very small amounts of labeled data.

8. Conclusion

Oliver et al. [4] mention: “Pseudo-labeling is a simple heuristic which is widely used in practice, likely because of its simplicity and generality” and as we’ve seen it provides a nice way to learn about Semi-Supervised Learning.

Over the last 2–3 years, Semi-Supervised Learning for Image classification has seen some incredible improvements. Unsupervised Data Augmentation (Xie et al (2019) [5]) has achieved 97.3% on CIFAR-10 with just 4000 Labels. To put that into perspective, DenseNet (Huang et al (2016) [6]) achieved 96.54% on the complete CIFAR-10 dataset in 2016.

It’s really interesting to see how the Machine Learning and Data Science community is moving towards algorithms that either uses less labeled data (like Semi-Supervised Learning, Zero/Few Shot Learning) or smaller datasets altogether (like Transfer Learning). Personally, I believe these developments are critical if we truly want to democratize Artificial Intelligence for all.

If you have any questions feel free to connect with me. I hope you enjoyed!

Github Repo:

https://github.com/anirudhshenoy/pseudo_labeling_small_datasets

Dataset: <https://www.kaggle.com/oddrational/mnist-in-csv>

References:

1. Dong-Hyun Lee. “Pseudo-Label : The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks” ICML 2013 Workshop : Challenges in Representation Learning (WREPL), Atlanta, Georgia, USA, 2013 (http://deeplearning.net/wp-content/uploads/2013/03/pseudo_label_final.pdf)
2. Eric Arazo, Diego Ortego, Paul Albert, Noel E. O’Connor, Kevin McGuinness. “Pseudo-Labeling and Confirmation Bias in Deep Semi-Supervised Learning” (<https://arxiv.org/abs/1908.02983>)
3. David Berthelot, Nicholas Carlini, Ian Goodfellow, Nicolas Papernot, Avital Oliver, Colin Raffel. “MixMatch: A Holistic Approach to Semi-Supervised Learning” (<https://arxiv.org/abs/1905.02249>)
4. “Avital Oliver, Augustus Odena, Colin Raffel, Ekin D. Cubuk, Ian J. Goodfellow. “Realistic Evaluation of Deep Semi-Supervised Learning Algorithms” (<https://arxiv.org/abs/1804.09170>)
5. Qizhe Xie, Zihang Dai, Eduard Hovy, Minh-Thang Luong, Quoc V. Le. “Unsupervised Data Augmentation for Consistency Training” (<https://arxiv.org/abs/1904.12848>)
6. Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger. “Densely Connected Convolutional Networks”


(<https://arxiv.org/abs/1608.06993>)

7. https://github.com/peimengsui/semi_supervised_mnist
8. <https://www.analyticsvidhya.com/blog/2017/09/pseudo-labelling-semi-supervised-learning-technique/>
9. <https://www.quora.com/Why-is-unsupervised-learning-important>
10. <https://www.wired.com/2014/08/deep-learning-yann-lecun/>

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

 Get this newsletter