Finetuning and Domain Adaptation

Antonio Carta

antonio.carta@unipi.it

Plan for Today

- what is finetuning
- how does it work: practical tips, transferability
- domain adaptation
- Test-time adaptation

Transfer learning be like

https://twitter.com/miniapeur/status/1606689673396359168/photo/1

Problem Definition and Motivations

Definition – Transfer Learning (TL)

- T_b a task, such as image classification of plants
- D_b a dataset sampled from T_b
- θ_b parameters of a DNN after training on D_b

Def – Transfer Learning:
Solve target task T_b after solving source task(s) T_a
by transferring knowledge learned from T_a

D_a is not available during TL

OBSERVATION: you can solve Multi-Task Learning (MTL) with TL methods but not viceversa. Not having access to D_a is a hard constraint.

Typical Setting

- D_a is very large
- D_b may be small
- We don’t have D_a (e.g. pretrained model from private company)
- We don’t care about solving T_a and T_b jointly
- Example: pretrain on ImageNet -> TL on specialized domain

Pretrained Models

Where do you get the pre-trained parameters?

- Pretrained models are available
  - e.g. ImageNet classification model
  - often available online (e.g. Huggingface)
- Models trained on large language corpora for NLP
- Whatever large, diverse dataset you might have
- Often these models are trained on different tasks:
  - See self-supervised lecture
  - Example: masked language modeling

Multi-Task Learning vs Transfer Learning

MTL: we have all the data at the same time

- we have multiple tasks
- often the tasks have a similar size/complexity

TL: we have only T_b

- Only two tasks T_a, T_b
- usually $D_b \ll D_a$

Finetuning and Transferability

2D Toy Experiment

- how would you split the data with hyperplanes?
- Do you think your split generalizes to new domains?
- can we even tell if our solution generalizes?

"Direct Transfer of Learned Information Among Neural Networks", L. Y. Pratt et al, AAAI 1991

Transfer Latent Features

- Better solution with DNN: reuse latent representations
- you may have to change the classification hyperplanes completely, but the latent features may still be helpful to solve related tasks
- **ASSUMPTION**: the tasks are related -> discriminative features learned for T_a are helpful for T_b
  - When does this assumption hold?

Finetuning

- Finetuning: SGD on D_b, starting from θ_a

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}\left(\theta, \mathcal{D}_a\right)$$

- SGD starts from pretrained model θ_a
- θ_b finetuned model
- D_b new data

Design Choices

Optional popular choices:

- **epochs**: usually less iterations/epochs than training from scratch
  - fast adaptation to similar tasks
  - avoids overfitting small datasets
- **learning rate**: α new learning rate, often smaller
- **weight decay**: may be set to 0
- **freezing**: small lr or freeze for early layers
- **reinit**: random reinit for last layers
- **Warm Start**: train only the last layer, then finetune everything

Finetuning – Warm Start

- start from a pretrained model θ_a
- freeze everything except the classifier
- randomly initialize the classifier
- finetune the classifier
- unfreeze all the parameters
- finetune everything

RATIONALE: the randomly initialized classifier may have large gradients, which result in large changes in the DNN.

- Warm start helps to reduce “forgetting” of the representations
- not always the best choice

Pytorch – different learning rates

- In pyorch, we can split model parameters into parameter group
- Each group has its own dictionary
  - The «params» key contains the parameters
  - The others contain group-specific parameters of the optimizer
  - Unspecified parameters will be set to the global default

```
optim.SGD([
    {'params': model.base.parameters(),
                'lr': 1e-2},
    {'params': model.classifier.parameters()}
], lr=1e-3, momentum=0.9)

optim.SGD([
    {'params': model.base.named_parameters(),
                'lr': 1e-2},
    {'params': model.classifier.named_parameters()}
], lr=1e-3, momentum=0.9)
```

Pytorch – reset classifier

- We can reset the classifier by creating a new Linear layer
- Always be careful about the initialization
  - Bias can be set to zero
  - Weights to a random distribution with small standard deviation
  - rationale: we want the activations to be approximately normal. The common initializations ensure this via a proper standard deviation of the weights

```
in_f = model.classifier.in_features
model.classifier = torch.nn.Linear(
        in_f, num_classes)
torch.nn.init.normal_(model.classifier.weight,
                      std=0.02)
torch.nn.init.zeros_(model.classifier.bias)
```

Pytorch – disabling backpropagation

- Each operation in pytorch has a forward and backward operation
  - Forward: given input, compute output
  - Backward: given output and state, compute gradient w.r.t. input
- Every computation in pytorch creates a computational graph, unless we disable it with the torch.zero_grad context manager
- The computational graph is visited backward to compute the gradient
- Disabling the gradient computation saves memory, because pytorch does not need to keep the computational graph

```
x = torch.tensor([1.], requires_grad=True)

with torch.no_grad():
    y = x * 2
# here y.requires_grad == False

@torch.no_grad()
def doubler(x):
    return x * 2
z = doubler(x)
# here z.requires_grad == False
```

Pytorch – freeze layers

- To compute the SGD on a Module we need two things
  - requires_grad = True (this is the default for Modules)
  - The parameter must be passed to the optimizer
- To freeze the layer we can
  - Set requires_grad = False
  - Not pass the Module to the optimizer

```
# freeze backbone, train head
for p in model.backbone.parameters():
    p.requires_grad_(False)

opt = torch.optim.AdamW(model.classifier.parameters(),
                        lr=1e-3, weight_decay=1e-2)
```

How transferable are learned features?

- we know early layers learn Gabor filters. These are generally useful for a large family of tasks
- is it true also for deeper layers?
- **INTUITION**: low layer are general feature extractor, high layers are task-specific

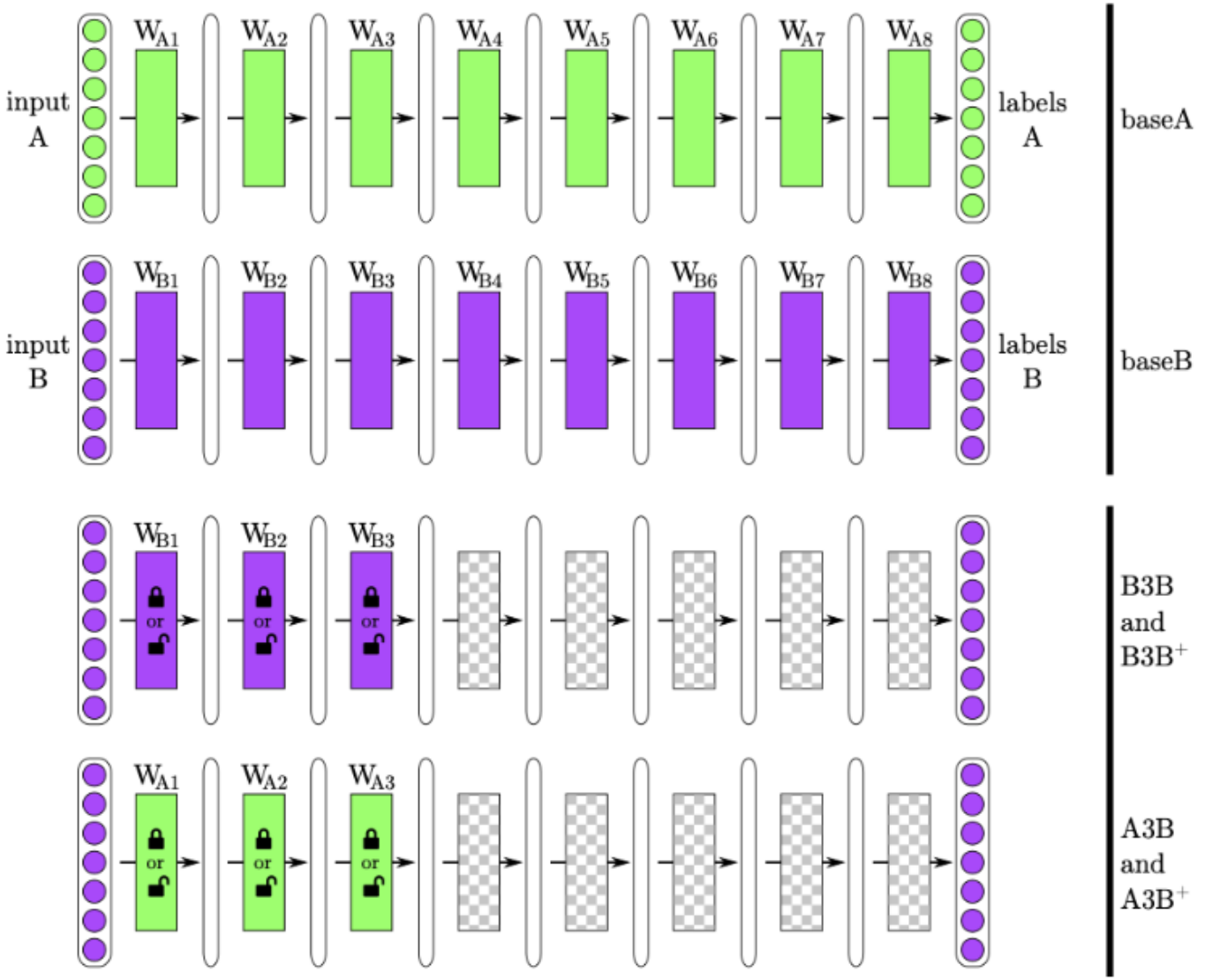
A Simple Transferability Experiment

- **Data**: two ImageNet splits A and B
- **self-transfer**: network trained on A and finetuned on A
- **transfer**: network trained on A and finetuned on B
- **training**: share first k layers, others are randomly initialized. Shared layers are frozen or finetuned (+ symbol in the plots)

Yosinski et al. "How Transferable Are Features in Deep Neural Networks?" NIPS'14

Transferability in ImageNet

- **Split** Imagenet into 2 sets of 500 classes: A and B
- "**Lock**" different sets of layers/representations & randomly initialize upper remaining layers
- Alternatively: **continue training/fine-tuning** transferred layers



Yosinski et al. "How Transferable Are Features in Deep Neural Networks?" NIPS'14

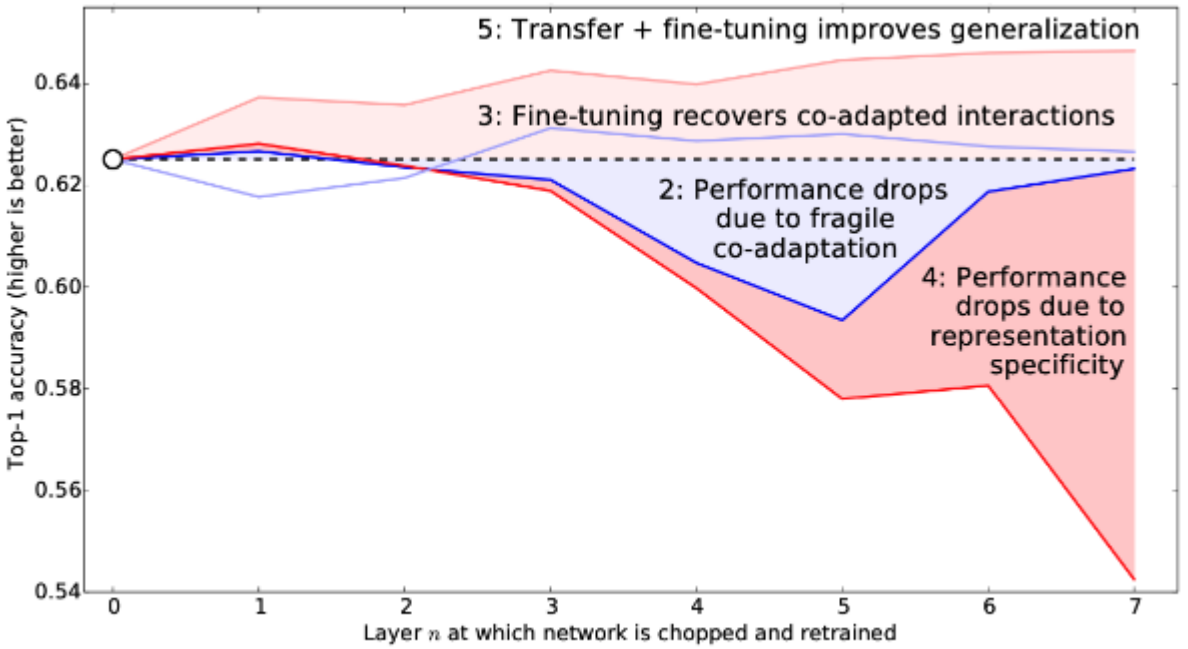
Results

2. B-B: copied from B and frozen + random rest trained on B

3. B-B+: copied features are allowed to adapt/fine-tune

4. A-B: transfer from A to B with frozen layers

5. A-B+: transferring + fine-tuning from A to B



Yosinski et al. "How Transferable Are Features in Deep Neural Networks?" NIPS'14

Size of the Pretraining Dataset Matters

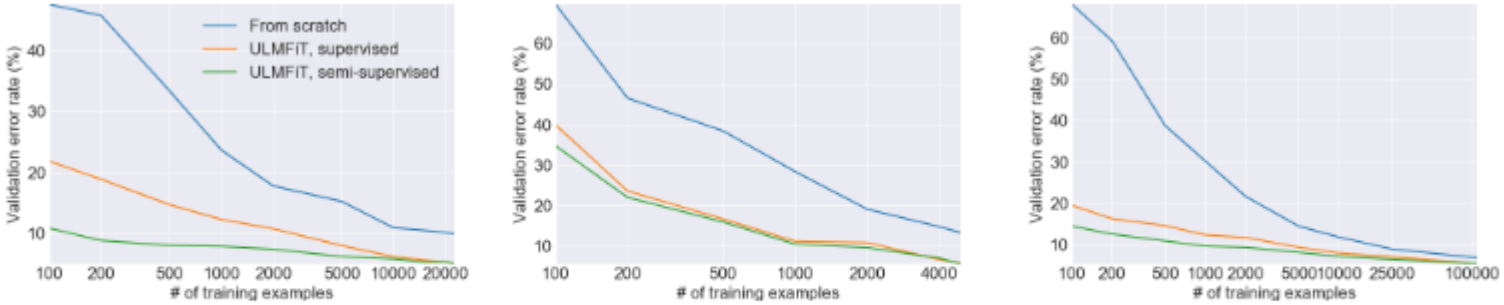


Figure 3: Validation error rates for supervised and semi-supervised ULMFiT vs. training from scratch with different numbers of training examples on IMDb, TREC-6, and AG (from left to right).

Domain Matters

- if you change the domain too much transfer may not work anymore
- examples: for some problems, low-level features don't matter. For others, they are critical
  - satellite images
  - head shots
  - natural images
  - x-ray medical images

"Meta-learning Convolutional Neural Architectures for Multi-target Concrete Defect Classification with the Concrete Defect Bridge Image Dataset", Mundt et al, CVPR 2019
"Material Recognition in the Wild with the Materials in Context Database, CVPR 2015"

DNN/CNN Texture Bias

(a) Texture image

81.4%	**Indian elephant**
10.3%	indri
8.2%	black swan

(b) Content image

71.1%	**tabby cat**
17.3%	grey fox
3.3%	Siamese cat

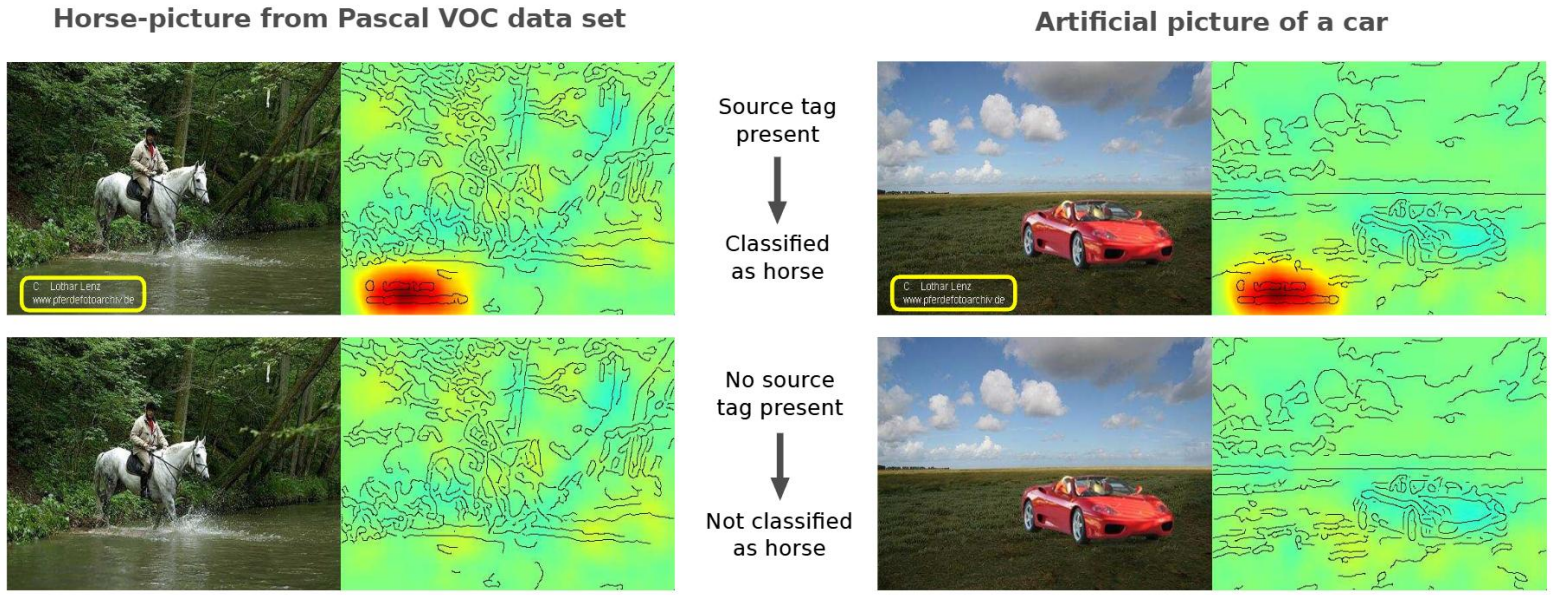
(c) Texture-shape cue conflict

63.9%	**Indian elephant**
26.4%	indri
9.6%	black swan

ImageNet-trained CNNS are biased towards texture", Geirhos et al, ICLR 2019

Confounders

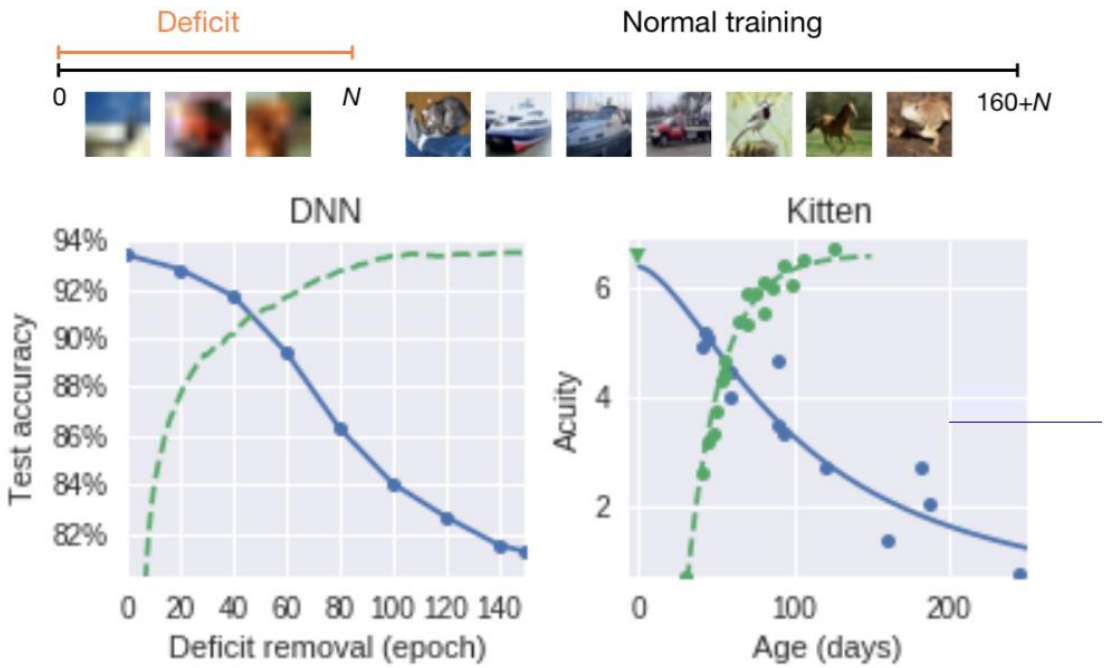
- Confounders and spurious correlations may also hurt TL performance
- Example:
  - what is the difference between house dogs and sled dogs?
  - DNN answer: the snow background



Unmasking Clever Hans Predictors", Lapuschkin et al, Nature Communications 2019

Critical Learning Periods in DNN

- Early stages of training are fundamental to achieve good performance
- Bad training conditions (less data, more noise, …) in the first epochs are not recoverable later during training
- The same result holds in neuroscience
- In the figure:
  - Left: DNN trained in the first epochs with heavily blurred images for the first x epochs
  - Right: cats with a covered eye in the first x days



A. Achille et al. "Critical Learning Periods in Deep Neural Networks", ICLR 2019

Domain Adaptation

Task vs Domain

- task: $\mathcal{T}_i \triangleq \{p_i(\mathbf{x}), p_i(\mathbf{y} \mid \mathbf{x}), \mathcal{L}_i\}$
- domain: $d_i \triangleq \{p_i(\mathbf{x}), p(\mathbf{y} \mid \mathbf{x}), \mathcal{L}\}$

in practice, given a task, a domain is a subset of the task.
Example:

- image classification of animals
- domains:
  - different environments: jungle, savannah, …
  - different images: distant images, close images, high/low res, …

Terminology

- **Source Domain:** the data distribution on which the model is trained using labeled examples
- **Target Domain**: the data distribution on which a model pre-trained on a different domain is used to perform a similar task
- **Domain Translation**: the problem of finding a meaningful correspondence between two domains
- **Domain Shift**: a change in the statistical distribution of data between different domains

Domain Adaptation Problem

- **Domain Adaptation** is a transfer learning problem where we have with access to target domain data during training.
- **Unsupervised Domain Adaptation**: unlabeled target domain data
- **Semi-supervised domain adaptation**: unlabeled data and a small labeled subset
- **Supervised domain adaptation**: labeled target domain data

Assumptions

- Source and target are different domains but closely related
- There exists a single hypothesis (model/DNN) with low error on both source and target data
  - in transfer learning the source and target task can be much more different
- the shift from source to target is a form of virtual drift

Unsupervised Domain Adaptation

- We focus on **Unsupervised Domain Adaptation**: unlabeled target domain data
- The goal is to adapt the model to classify the new domain
- The challenge is that we don’t have the labels for the new domain
- Methods are heuristics or based on strong assumptions about the domain shift

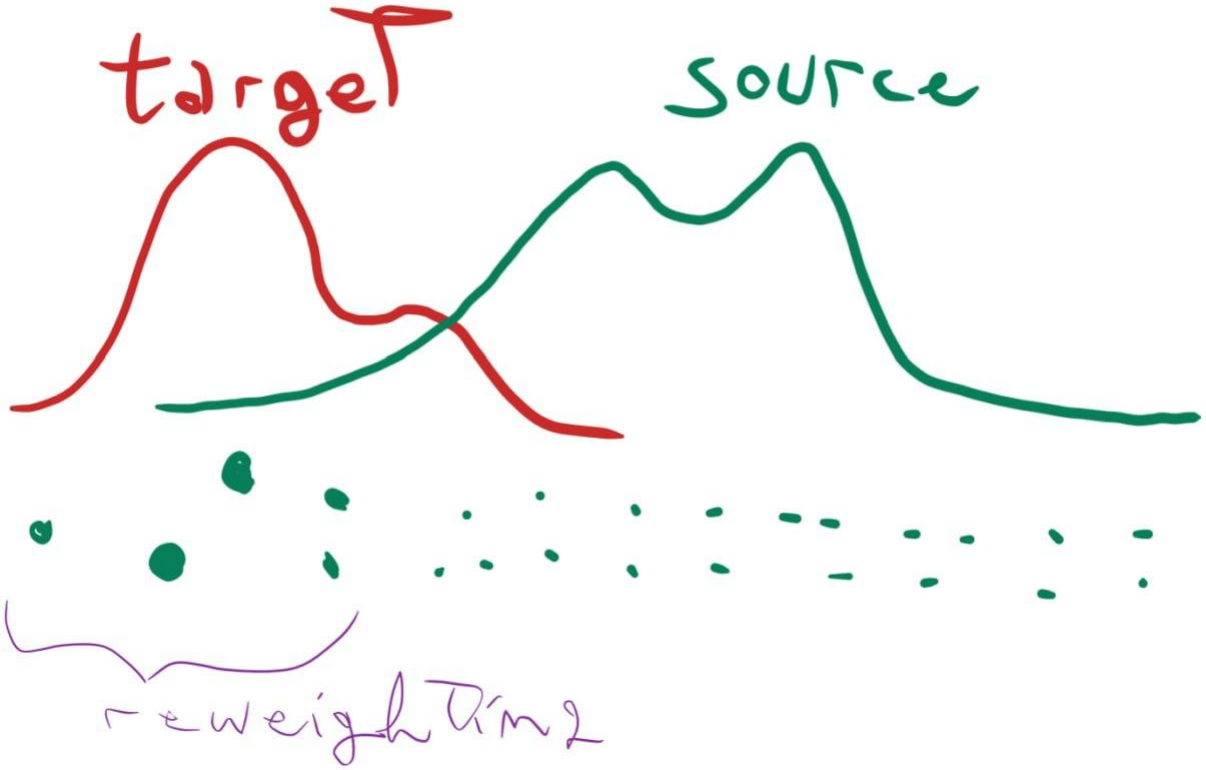
Domain Adaptation Methods

- **Data reweighting**: importance sampling
- **Feature Alignment**: ensures that source and target features are aligned
- **Domain Translation**: learn “translation” functions from source to target (or viceversa). A generalization of the concept of language translation.

Domain Bias

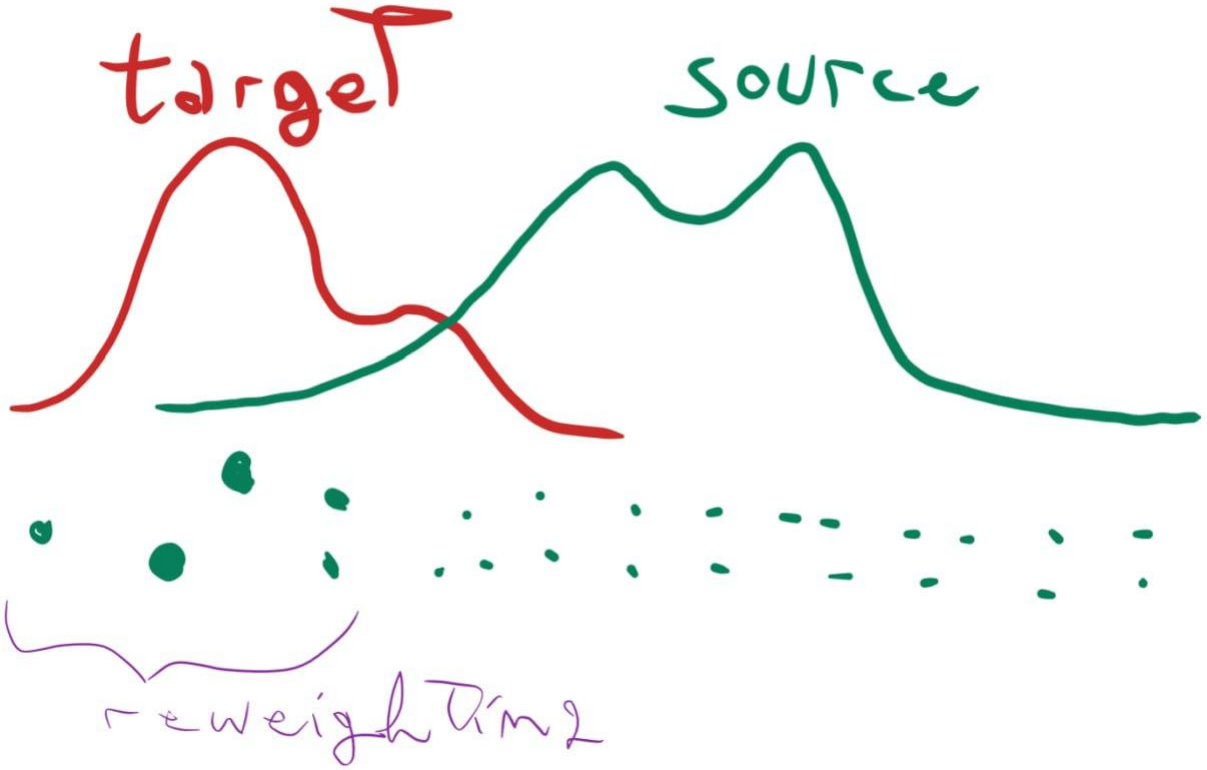
- p_s source distribution
- p_T target distribution
- model trained on $p_s(x, y)$ ignores samples from $p_T(x, y)$

How can we mitigate this issue? we can use the (unlabeled) target data



Sample Selection Bias

- **REMEMBER**: selection bias is a form of virtual drift!
- this is an imbalance problem
- **IDEA**: weigh more samples with low source probability (p_S) and high target probability (p_T)



Source and Target Error

- Error on source: $E_{p_S(x,y)}[\mathcal{L}(x, y, \theta)]$
- Error on target: $E_{p_T(x,y)}[\mathcal{L}(x, y, \theta)]$

- **Derivation**:

$$\begin{aligned}\mathbb{E}_{p_T(x,y)}[\mathcal{L}(x, y, \theta)] &= \int p_T(x, y)\mathcal{L}(x, y, \theta)dxdy \\ &= \int p_T(x, y)\frac{p_S(x, y)}{p_S(x, y)}\mathcal{L}(x, y, \theta)dxdy \\ &= \mathbb{E}_{p_S(x,y)}\left[\frac{p_T(x, y)}{p_S(x, y)}\mathcal{L}(x, y, \theta)\right]\end{aligned}$$

- **solution**: minimize error on target domain by weighing source data by $p_T(x,y) / p_S(x,y)$
- **problem**: we need a generative model for the joint distributions p_T and p_S

Importance Sampling (IS)

- $p(y|x)$ is domain-independent -> we can ignore it
- $p(x)$: Apply Bayes rule to the importance sampling coefficient

$$\frac{p_T(x)}{p_S(x)} = \frac{p(x|\text{ target })}{p(x|\text{ source })} = \frac{p(\text{ target }|x)p(\text{ source })}{p(\text{ source }|x)p(\text{ target })}$$

- $p(source \mid x)$ is a binary domain classifier
- $p(source)/p(target)$ is a constant term that we can remove without changing the optimal solution

Importance Sampling Algorithm

training algorithm:

- train domain classifier $p(source \mid x; \theta)$ to classify source/target
- reweight samples by $w_i = \frac{1 - p(source|x_i;\theta))}{p(source|x_i;\theta))}$
- Minimize $w_i L(x_i, w_i, \theta)$

IS Assumption

$$p_T(x, y) \neq 0 \implies p_S(x, y) \neq 0$$

- informally, the source domain contains the target domain
- approximately true if you go from a general domain (ImageNet) to a speficic one (birds classification)
- probably false if you switch from one specialized domain to another (birds→fish)

Test-Time Adaptation (TTA)

- TTA is the online version of unsupervised domain adaptation
- We want to adapt the model to the target domain during the inference time
- We need quick adaptation: the test data is seen online (prequential evaluation)
- We need stability: noisy updates will result in an unreliable model

Test Entropy Minimization (TENT)

- $\hat{y}$ model predictions
- Entropy $\mathrm{H}(\hat{y}) = -\sum_c p(\hat{y}_c)\log p(\hat{y}_c)$
  - c are the classes
- Entropy is minimized when the model is more confident
- We minimize the average entropy of a minibatch
  - Reduce noise and improves stability
  - The solution for a single example would be to set the most confident class to probability 1

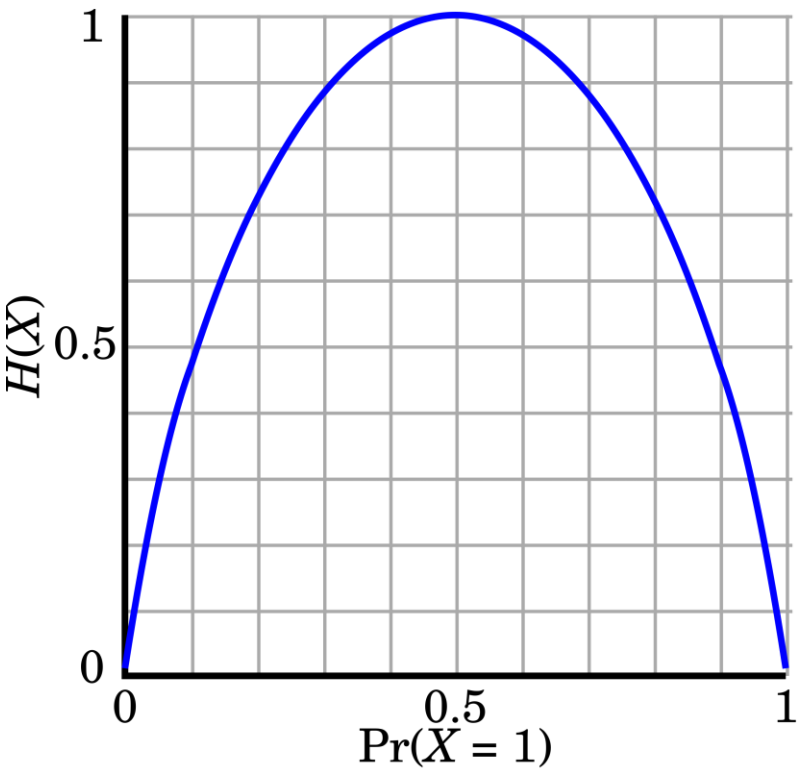


Image: wikimedia

TENT – Feature Modulation

- We want to avoid training the whole network parameters
  - Inefficient and unstable
- TENT only adapts «feature modulators», affine layers that rescale the activations:
  - This is the usual BatchNorm layer
  - $E[x]$ and $Var[x]$ are the mean and variance estimated online for the data
  - γ, β are learnable parameters

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

BatchNorm1d — PyTorch 2.10 documentation

Conclusion

Take-Home Messages

- sometimes, we don’t really care about preserving the performance on the old task
- finetuning/domain adaptation allows to quickly learn new task/domains
- Knowing whether there will be forward transfer is never intuitive. Test your assumptions.

References

- Slides should be enough
- CS 330 slides http://cs330.stanford.edu/
- You can check the papers in the footnotes for more info

Next-Lecture

Multi-Task learning

- Definition
- Design choices
- challenges