Transfer Learning

Christoph H. Lampert
IST Austria (Institute of Science and Technology Austria), Vienna

IWR School "A Crash Course in Machine Learning
with Applications in Natural- and Life Sciences (ML4Nature)"

September 25, 2019 | Heidelberg, Germany
(with material by Pierre Geurts, Gilles Louppe, Louis Wehenkel)

IST Austria (Institute of Science and Technology Austria)

About IST Austria

- publicly funded research institute
- located in outskirts of Vienna

Research at IST Austria

- curiosity-driven basic research
- focus on interdisciplinarity:
  - computer science, mathematics, neuroscience, physics, biology, chemistry

IST Austria Graduate School

- English language program
- 1 + 3 yr PhD program
- fully funded, no fees

More information: www.ist.ac.at, chl@ist.ac.at, or ask me during the break

Schedule

What is Transfer Learning?

Why Transfer Learning?

Asymmetric Transfer

Symmetric Transfer

Other(?)

Slides available at: `http://cvml.ist.ac.at`

What is Transfer Learning?

What is Transfer Learning?

What is Transfer Learning?

In the deep learning community sometimes used to mean:

> "Training a deep learning model on one dataset and then fine-tuning it on a different one."

What is Transfer Learning?

In the deep learning community sometimes used to mean:

~~"Training a deep learning model on one dataset and then fine-tuning it on a different one."~~

Better (more general) usage:

"Solving a machine learning task by making use of data or information from another learning task."

What is Transfer Learning?

What kind of information or data?

Example: transfering parameters

We want to train a model for translating Portugese to English.
For this, we reuse some parameters of an existing model for translating Spanish to English.

Example: transfering data points

We want to train a model that detects zebras in videos.
For this, we use images that show horses in addition to images that actually show zebras.

Example: transfering high-level information

We want to train a model that marks synapses in electron microscope images.
For this, we use a network architecture that proved useful for natural images.

Previous example were asymmetric:

Asymmetric transfer

There's one target task we care about, and to do so we make use other data or information of another task.

Previous example were asymmetric:

Asymmetric transfer

There's one target task we care about, and to do so we make use other data or information of another task.

Symmetric settings are also possible:

Symmetric transfer

There's several tasks we care about. While learning all of them, they share information with each other.

Example: Multi-task learning

We want to train models for recognizing handwriting for many languages (e.g. 82 on Android). We train all models jointly instead of each of them separately.

Many transfer learning scenarios and methods:

- pretrained features
- model fine-tuning
- domain adaptation
- weakly-supervised learning
- strongly-supervised learning
- zero-shot learning
- multi-task learning
- learning to learn / meta-learning

It's important to understand not only how they work, but also when and why.

When and Why Transfer Learning?

Reminder:

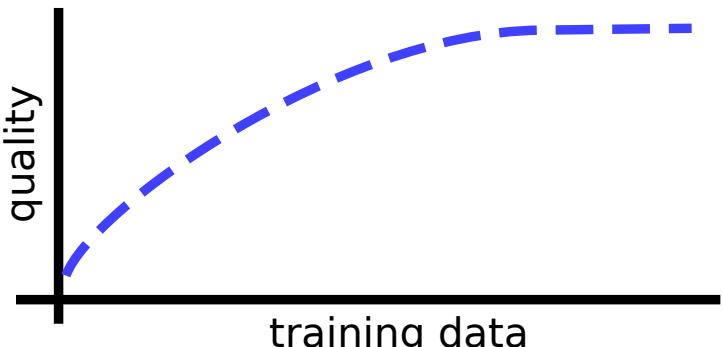
- input space: $\mathcal{X}$ e.g. color images of size 256×256: $\mathcal{X} = \mathbb{R}^{256 \times 256 \times 3}$
- output space: $\mathcal{Y}$ e.g. $\mathcal{Y} = \{\text{cat}, \text{dog}\}$
- goal: classifier $f : \mathcal{X} \to \mathcal{Y}$ e.g. $f_\theta(X)$, a neural network with parameters θ
- given: training data $(X_1, Y_1), \ldots, (X_N, Y_N)$ from the data distribution $p(X, Y)$
- typical method:

$$\min_\theta \mathcal{L}(\theta) \qquad \text{for} \quad \mathcal{L}(\theta) = \sum_{i=1}^{N} \text{Loss}\big(Y_i, f_\theta(X_i)\big) + \text{regularization or tricks}$$

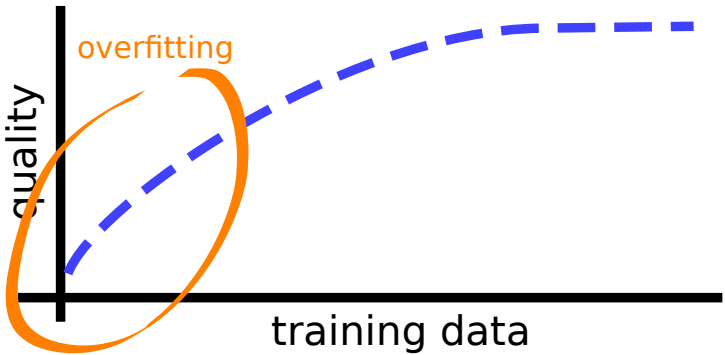
Observation:

- We know that this is the optimal strategy, if we have enough data!

Main problem: powerful nonlinear models, e.g. deep networks, need a lot of training data to give good results.

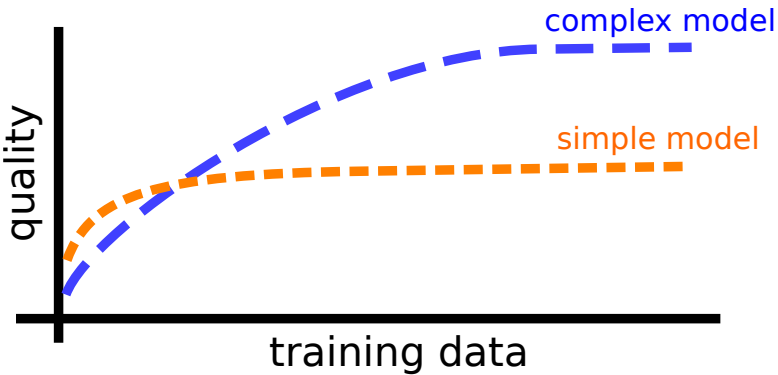


Main problem: powerful nonlinear models, e.g. deep networks, need a lot of training data to give good results.



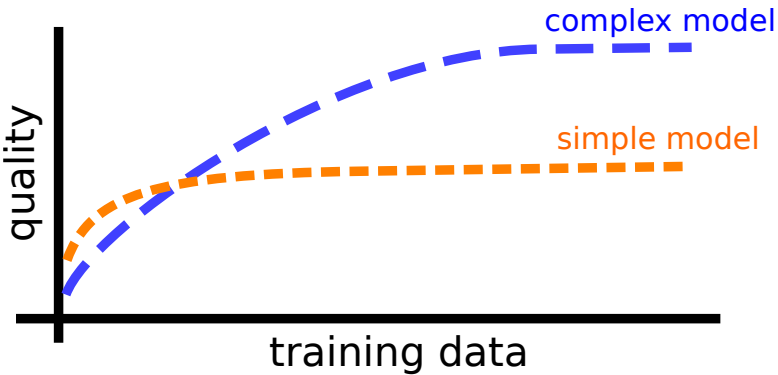
- too little data: overfitting, bad prediction quality

Main problem: powerful nonlinear models, e.g. deep networks, need a lot of training data to give good results.



- simple model, e.g. linear: less overfitting, but low asymptote for large data

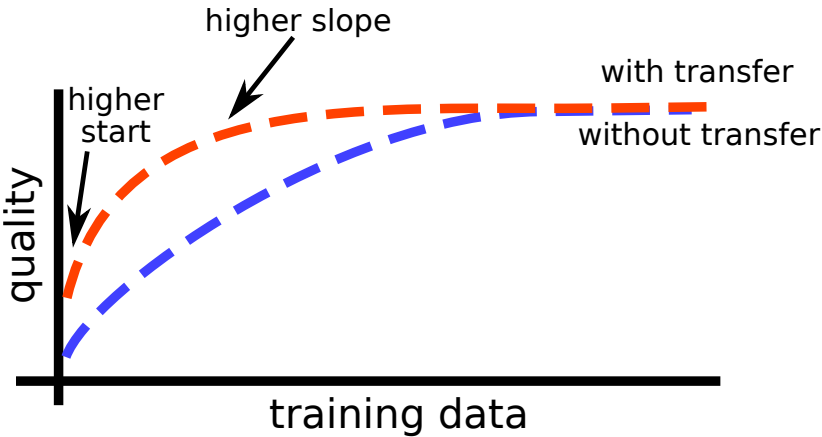
Main problem: powerful nonlinear models, e.g. deep networks, need a lot of training data to give good results.



- simple model, e.g. linear: less overfitting, but low asymptote for large data

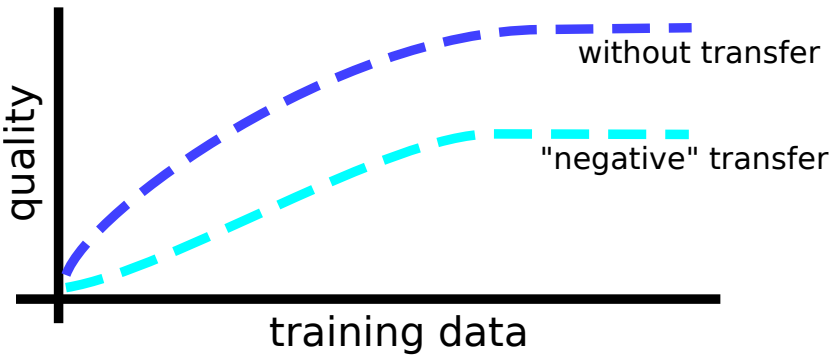
Transfer learning tries to achieve the best of both worlds.

Main problem: powerful nonlinear models, e.g. deep networks, need a lot of training data to give good results.



- ideal result of transfer learning: good results even with little data

Main problem: powerful nonlinear models, e.g. deep networks, need a lot of training data to give good results.



- beware: used incorrectly, transfer learning can have negative effects

Reminder:

- input space: $\mathcal{X}$ e.g. color images of size 256×256: $\mathcal{X} = \mathbb{R}^{256 \times 256 \times 3}$
- output space: $\mathcal{Y}$ e.g. $\mathcal{Y} = \{\text{cat}, \text{dog}\}$
- goal: classifier $f : \mathcal{X} \to \mathcal{Y}$ e.g. $f_\theta(X)$, a neural network with parameters θ
- given: training data $(X_1, Y_1), \ldots, (X_N, Y_N)$ from the data distribution $p(X, Y)$
- typical method:

$$\min_\theta \mathcal{L}(\theta) \qquad \text{for} \quad \mathcal{L}(\theta) = \sum_{i=1}^{N} \text{Loss}\big(Y_i, f_\theta(X_i)\big) \qquad + \text{regularization or tricks}$$

Transfer learning:

- given: additional data, but not from the correct data distribution $p(X, Y)$

or

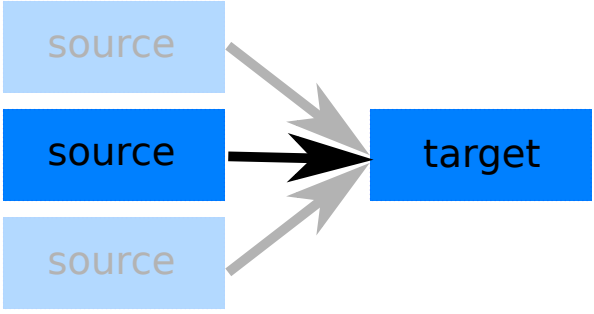
- given: a model g_η that solves a different task

Asymmetric Transfer

General idea: we have one task that we actually want to solve (called "target task"), but we do not have not enough training data for it. We use information or data from one or more other tasks (called "source tasks") for support.

Situations/Methods:

- pretrained features
- model fine-tuning
- domain adaptation
- weakly-supervised learning
- strongly-supervised learning
- zero-shot learning
- ...



Part I

Running Example: pigeon detector

- **goal:** scare away the pigeons from our apartment's terrace
- already available: remote-controlled water pistol
- missing: a model that detects if there's a pigeon in an image
- training data: tens or hundreds of examples (collected myself), not millions

Observations:

- not enough to simply train a ConvNet
- probably enough to train a linear model, but:
  linear models do not work as well as deep ones

Running Example: pigeon detector

- **goal:** scare away the pigeons from our apartment's terrace
- already available: remote-controlled water pistol
- missing: a model that detects if there's a pigeon in an image
- training data: tens or hundreds of examples (collected myself), not millions

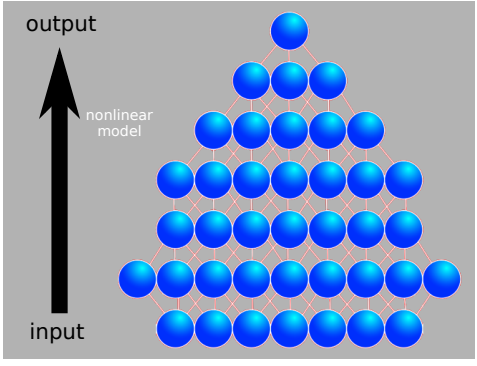
Observations:

- not enough to simply train a ConvNet
- probably enough to train a linear model, but:
  ~~linear models do not work as well as deep ones~~ **not true!**

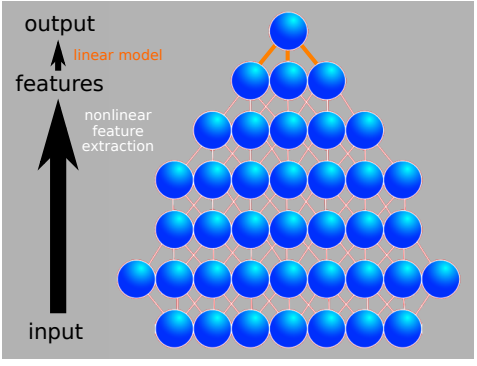
  linear models on raw inputs do not work as well as deep ones.
  But, with the right features, a linear model is as good as a deep one

After training, any deep network model **is** a linear model on top of a feature extractor.

$$f_\theta(X) = \overbrace{W_L\sigma(W_{L-1}\sigma(W_{L-2}\sigma(\cdots\sigma(W_1X))))}^{\text{deep network}} \quad \text{for } \theta = (W_1, W_2, \ldots, W_L)$$
$$= \underbrace{\phi(X)^\top\beta}_{\text{linear model}} \qquad \text{for } \phi(X) = \sigma(W_{L-1}\sigma(W_{L-2}\sigma(\cdots\sigma(W_1X)))) \text{ and } \beta = W_L$$



deep network view



linear model after features extraction view

After training, any deep network model **is** a linear model on top of a feature extractor.

So, what's the difference?

- for classical linear models, a feature map $\phi : \mathcal{X} \to \mathbb{R}^F$ is designed manually
  - few free parameters, works with little data, but asymptotically not best performance
- in deep learning, the features and the linear model parameters are trained jointly
  - a lot of free parameters, needs a lot of data

After training, any deep network model **is** a linear model on top of a feature extractor.

So, what's the difference?

- for classical linear models, a feature map $\phi : \mathcal{X} \to \mathbb{R}^F$ is designed manually
  - few free parameters, works with little data, but asymptotically not best performance
- in deep learning, the features and the linear model parameters are trained jointly
  - a lot of free parameters, needs a lot of data
- **transfer learning idea**: learn the features but on other data

After training, any deep network model **is** a linear model on top of a feature extractor.

So, what's the difference?

- for classical linear models, a feature map $\phi : \mathcal{X} \to \mathbb{R}^F$ is designed manually
  - few free parameters, works with little data, but asymptotically not best performance
- in deep learning, the features and the linear model parameters are trained jointly
  - a lot of free parameters, needs a lot of data
- **transfer learning idea**: learn the features but on other data

Intuition:

- classifier parameters are crucial to get right, but
- features parameters are more tolerant, many different feature maps can be useful (even ConvNets with random weights in feature part can still work okay)
- empirically, features that work well for one task, often also work for related tasks

Asymmetric Transfer

Pretrained Features

Pretrained Features: Procedure with Feature Extraction

1) Network Preparation

- train a deep network on source data or download an already trained one, $f : \mathcal{X} \to \mathcal{Y}'$
  - input space must match target input $\mathcal{X}$
  - output space $\mathcal{Y}'$ can be different from target output $\mathcal{Y}$
- remove (or simply ignore) one or more layers from the end
- call the resulting map, $\phi : \mathcal{X} \to \mathbb{R}^F$ (feature map)

2) Feature Extraction

- for each training example $(X_i)_{i=1}^N$ of the target task
  - evaluate $\phi(X_i)$ and store the result as feature vector $\tilde{X}_i$

3) Actual Training

- train a linear model, e.g. logistic regression, on the data $(\tilde{X}_1, Y_1), \ldots, (\tilde{X}_N, Y_N)$.

Pretrained features: Procedure with Feature Extraction

```
import tensorflow as tf
import tensornets as nets

inputs = tf.placeholder(tf.float32, [None, 224, 224, 3])
outputs = tf.placeholder(tf.float32, [None, 1])

model = nets.ResNet50(inputs)
features = model.get_outputs()[-3] # remove classifier layers

all_phi = []
with tf.Session() as sess:
    nets.pretrained(model) # load parameters of pretrained model
    for X in targetdata:
        phiX = sess.run(features, {inputs: X}) # train linear model
        all_phi.append(phiX)

from sklearn.linear_model import LinearRegression
trained_model = LinearRegression().fit(phiX, Y)
```

1) Network Preparation

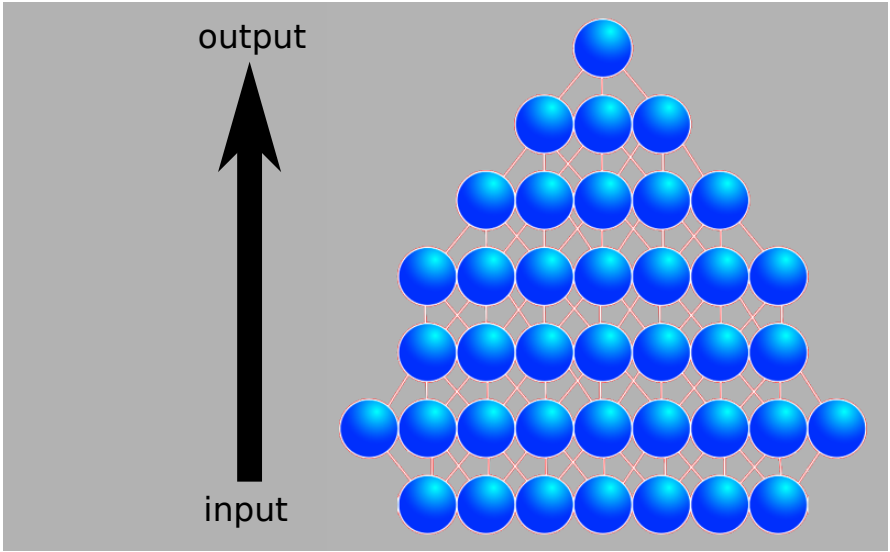
- train a deep network on source data or download an already trained one, $f : \mathcal{X} \to \mathcal{Y}'$
  - input space must match target input $\mathcal{X}$
  - output space $\mathcal{Y}'$ can be different from target output $\mathcal{Y}$
- remove (or simply ignore) one or more layers from the end
- call the resulting map, $\phi : \mathcal{X} \to \mathbb{R}^F$ (feature map)
- append a new linear classification layer with parameters $\beta \in \mathbb{R}^{F \times C}$, such that the network now compute $\tilde{f}(X; \beta) = \phi(X)^\top \beta \in \mathbb{R}^C$

2) Actual Training

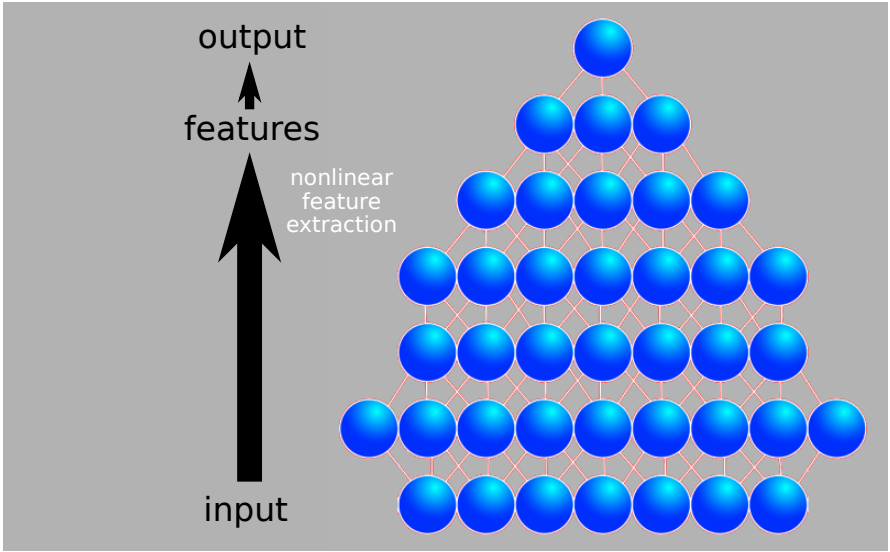
- train model by minimizing $\min_{\beta} \sum_{i=1}^{N} \text{Loss}\big(Y_i, \tilde{f}(X_i; \beta)\big)$,

  i.e. the linear model parameters change, the other network parameters are "frozen"

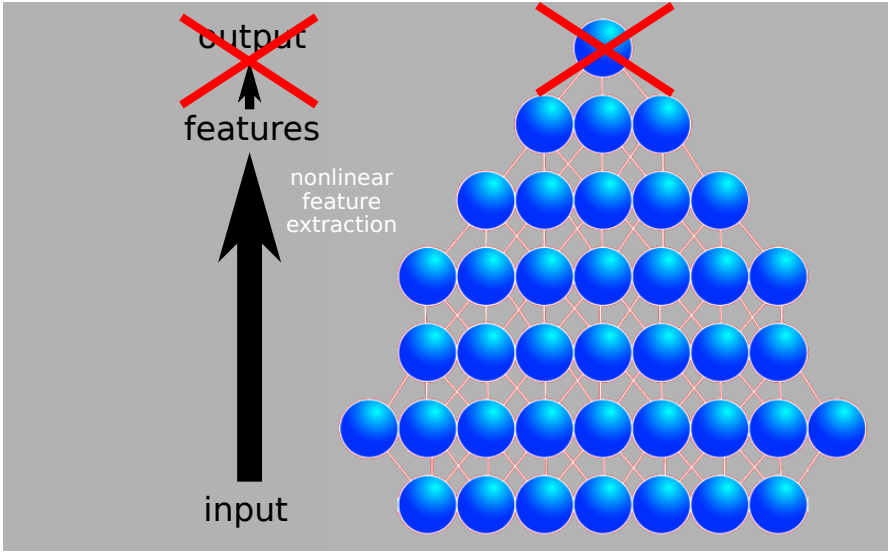
Network surgery:



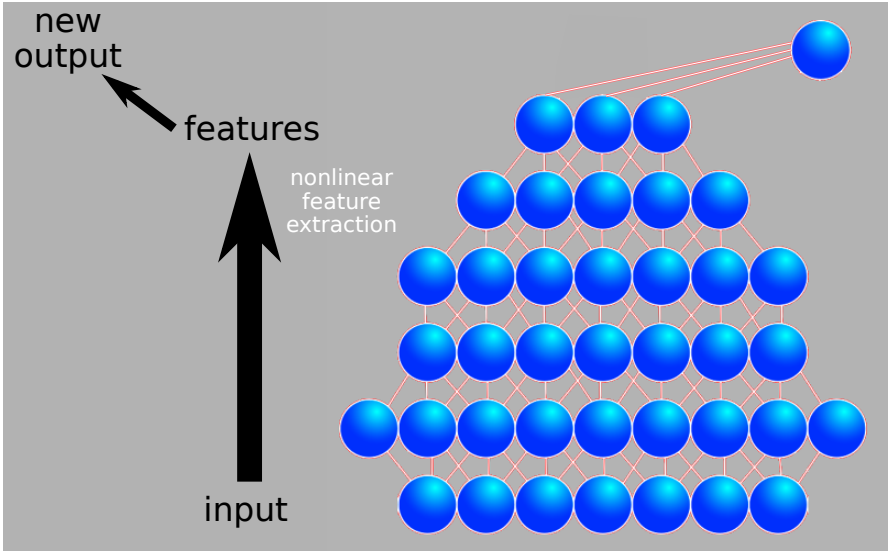
Network surgery:



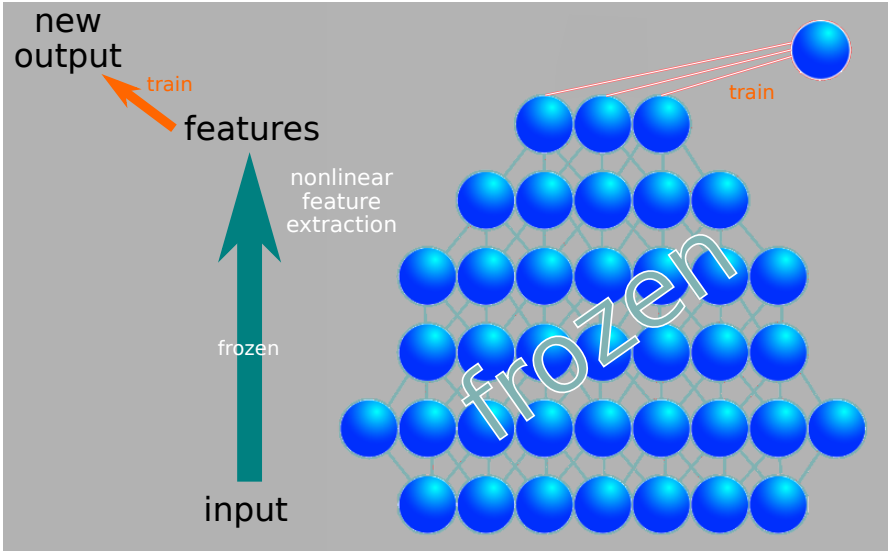
Network surgery:



Network surgery:



Network surgery:



ImageNet ILSVRC (2012) dataset: `http://image-net.org`

- 1.2 million images of 1000 object classes $\rightarrow$ enough to train large and deep ConvNets

Many popular architectures:

- general purpose: AlexNet (7 layers), VGG (18 layers), ResNet (18, 50, 101 layers), ...
- light-weight, e.g., for mobile devices: MobileNet, SqueezeNet, ...
- automatically generated: NASNetAlarge, ...

Many pretrained ConvNets publicly available online:

- Deep Learning Model Zoo: `https://modelzoo.co/`
- Tensornets: `https://github.com/taehoonlee/tensornets/`

AudioSet https://research.google.com/audioset/

- 2,084,320 human-labeled 10-second sound clips (from YouTube videos)
- annotated with 632 audio event classes

Pretrained networks:

- VGGish:
  https://github.com/tensorflow/models/tree/master/research/audioset
- model zoo: https://modelzoo.co/model/audioset

Pretrained features: Source code

```
import tensorflow as tf
import tensornets as nets

inputs = tf.placeholder(tf.float32, [None, 224, 224, 3])
outputs = tf.placeholder(tf.float32, [None, 1])

model = nets.ResNet50(inputs)
features = model.get_outputs()[-3] # remove classifier layers
beta = tf.Variable(tf.zeros(shape=(features.shape[1], 1), dtype=tf.float32))
predictions = tf.sigmoid(tf.matmul(features, beta))

loss = tf.losses.softmax_cross_entropy(outputs, predictions)
train = tf.train.AdamOptimizer().minimize(loss, var_list=[beta])

with tf.Session() as sess:
    nets.pretrained(model) # load parameters of pretrained model
    for (X, Y) in targetdata:
        sess.run(train, {inputs: X, outputs: Y}) # train linear model
```

Pretrained features: how good is it really?

CNN features off-the-shelf: an astounding baseline for recognition

A Sharif Razavian, H Azizpour, J Sullivan... - Proceedings of the ..., 2014 - cv-foundation.org

Recent results indicate that the generic descriptors extracted from the convolutional neural networks are very powerful. This paper adds to the mounting evidence that this is indeed the case. We report on a series of experiments conducted for different recognition tasks using the publicly available code and model of the OverFeat network which was trained to perform object classification on ILSVRC13. We use features extracted from the OverFeat network as a generic image representation to tackle the diverse range of recognition tasks of object ...

☆ 99 Cited by 3099 Related articles All 17 versions »

CVPR Workshop 2014

OverFeat network, trained on ImageNet, 4096-dimensional features:

- object categorization ✓
- fine-grained classification ✓
- attribute prediction ✓
- instance-retrieval (even for 3D objects) ✓
- ...

Take home message: Pretrained features

Pretrained features are easy to implement and surprisingly powerful. Try them first!

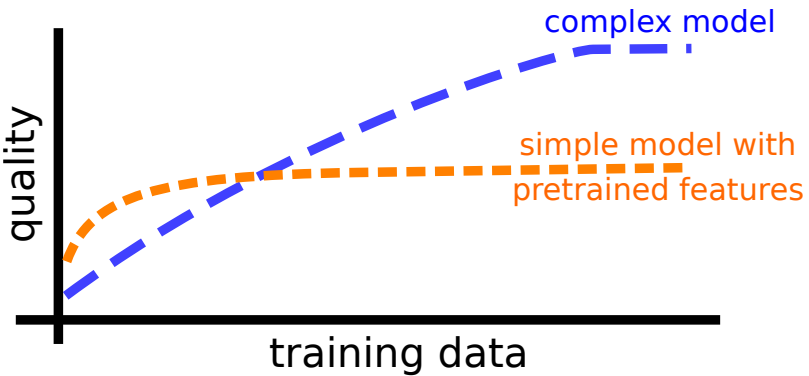
Pretrained features can make the system as easy and data-efficient as a linear model.

Given a related source task with enough data, quality is often better than with manually designing features.

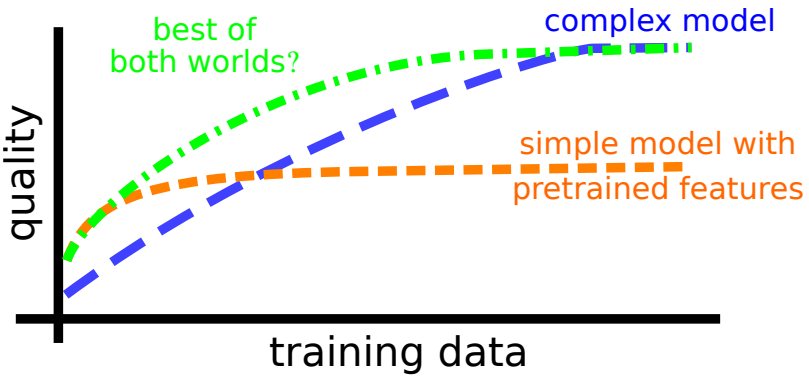
Other models can be used with pretrained features rather than just linear ones:

- if more target data is available, a non-linear model might work better,
- if even less target data is available, nearest neighbor classification might still work.

Can we do even better?



Can we do even better?



Asymmetric Transfer

Finetuning

Observation: Features from one task are good, but likely not optimal.

Task: Can we identify better features without having to completely train a deep network?

Finetuning

- 1) initialize network parameters from a pretrained network
- 2) train **all parameters** in the network, but just for a little bit so the parameters don't move far from their initial conditions:
  - train only for a few epochs, or
  - use a small learning rate

Finetuning

Observation: Features from one task are good, but likely not optimal.

Task: Can we identify better features without having to completely train a deep network?

Finetuning

- 1) initialize network parameters from a pretrained network
- 2) train **all parameters** in the network, but just for a little bit so the parameters don't move far from their initial conditions:
  - train only for a few epochs, or
  - use a small learning rate

Trade-off:

- no finetuning: original features, good but not perfect
- a little bit of finetuning: better features, better model
- too much finetuning: the model will forget its initialization $\rightarrow$ danger of overfitting

1) Network Preparation

- train a deep network on source data or download an already trained one, $f : \mathcal{X} \to \mathcal{Y}'$
  - input space must match target input $\mathcal{X}$
  - output space $\mathcal{Y}'$ can be different from target output $\mathcal{Y}$
- remove (or simply ignore) one or more layers from the end
- call the resulting map, $\phi_\theta : \mathcal{X} \to \mathbb{R}^F$ (feature map)
- append a new linear classification layer with parameters $\beta \in \mathbb{R}^{F \times C}$, such that the network now computes $\tilde{f}(X; \beta, \theta) = \phi_\theta(X)^\top \beta \in \mathbb{R}^C$

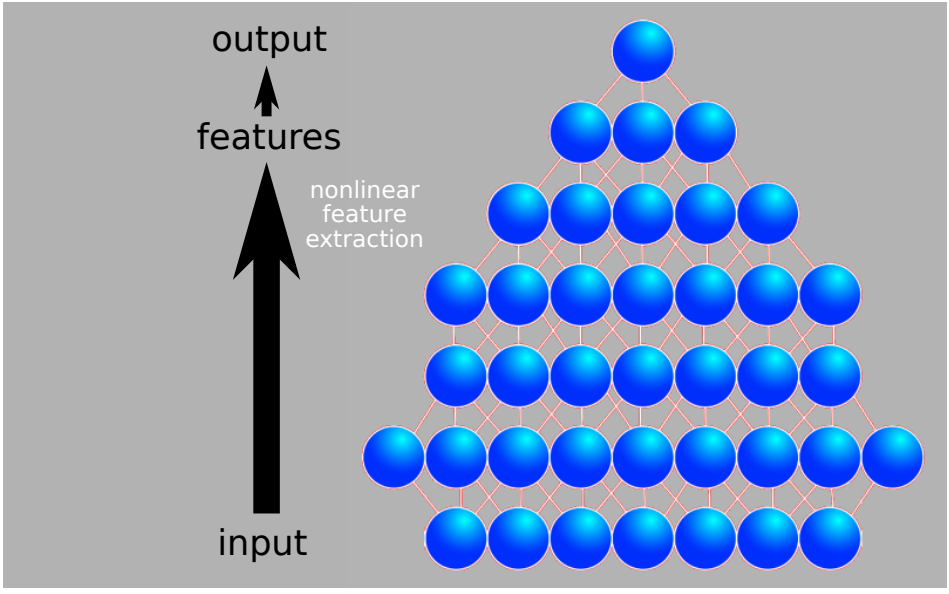
2) Actual Training

- train model by running a few steps of (stochastic) gradient descent on

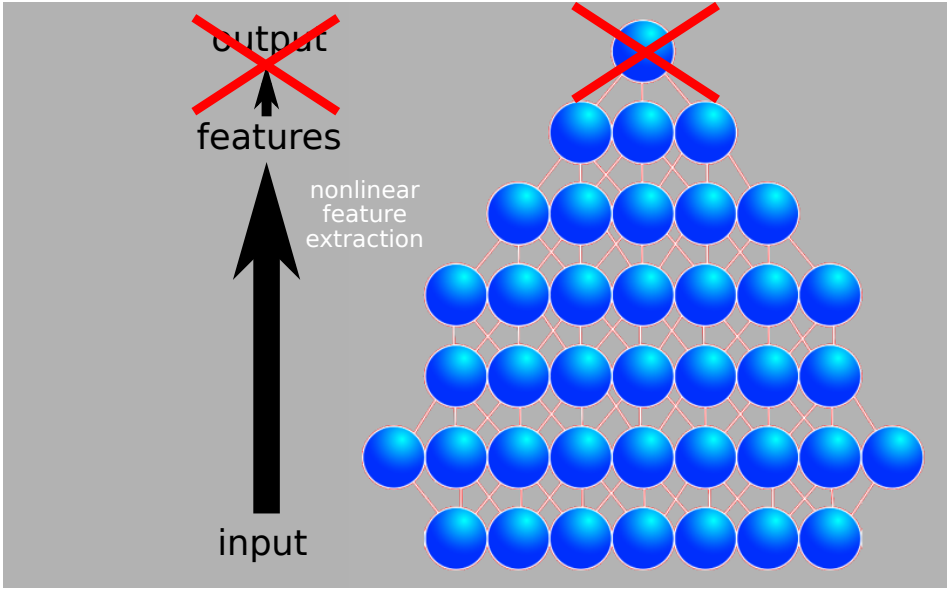
$$\min_{\beta,\theta} \sum_{i=1}^{N} \mathsf{Loss}\big(Y_i, \tilde{f}(X_i; \beta, \theta)\big),$$

i.e. all parameters are influenced, but only a little bit.

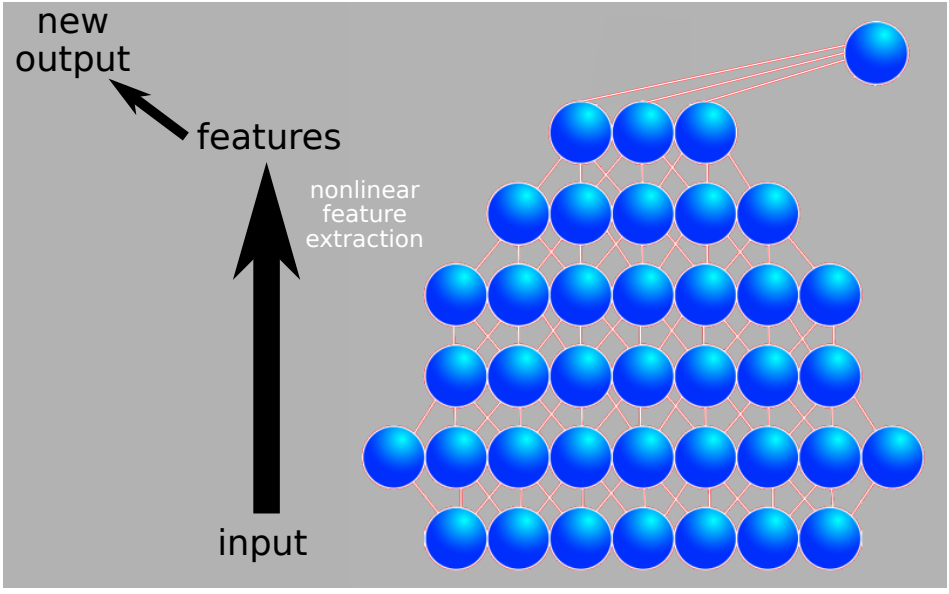
Pretrained Features: Procedure with Frozen Layers



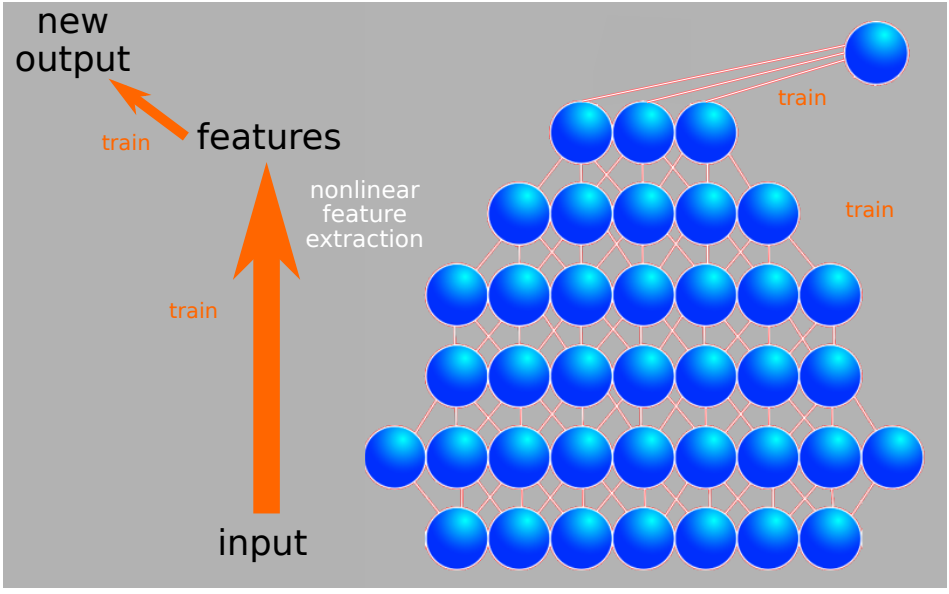
Pretrained Features: Procedure with Frozen Layers



Pretrained Features: Procedure with Frozen Layers



Pretrained Features: Procedure with Frozen Layers



Finetuning features: Source code

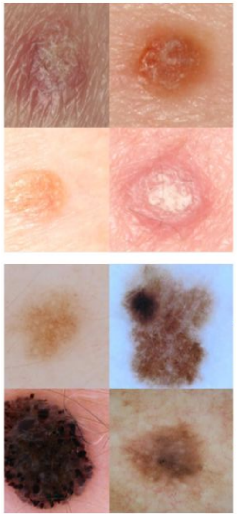
```
import tensorflow as tf
import tensornets as nets

inputs = tf.placeholder(tf.float32, [None, 224, 224, 3])
outputs = tf.placeholder(tf.float32, [None, 1])

model = nets.ResNet50(inputs, is_training=True)
features = model.get_outputs()[-3] # remove classifier layers
beta = tf.Variable(tf.zeros(shape=(features.shape[1], 1), dtype=tf.float32))
predictions = tf.matmul(features, beta)

loss = tf.losses.softmax_cross_entropy(outputs, predictions)
train = tf.train.AdamOptimizer(learning_rate=1e-5).minimize(loss)

with tf.Session() as sess:
    nets.pretrained(model) # load parameters of pretrained model
    for (x, y) in targetdata:
        sess.run(train, {inputs: x, outputs: y}) # train linear model
```

Dermatologist-level classification of skin cancer

- 48 layers ConvNet (Inception V3)
- pretrained on ImageNet
- fine-tuned on 129,450 medical images of 757 skin diseases
- simplified at prediction time to 3 or 9 classes

Final model performs on par or better than dermatologists:

- 3 classes: 72% vs. 66% accuracy
- 9 classes: 55% vs. 54% accuracy



(Esteva et al, Nature 2017)

Take home message: Finetuning

Finetuning is easy if you use a deep learning package anyway.

Finetuning often improves results over pretrained features. How much exactly depends on the task.

When finetuning one has to make sure not to train **too much**.

Finetuning and pretrained features are not mutually exclusive. One can freeze some layers and finetune others (e.g. typically at the last).

Flextuning [Royer, Lampert. under revision]

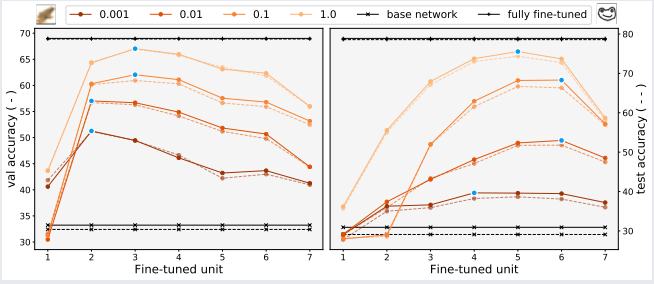
Situation:

- target task has same outputs as source task
- input characteristics might change, e.g. photo → line drawing

Idea: why train/finetune only the last layer(s)? why not early or intermediate ones?

Results:

- finetuning intermediate layer can work better than either finetuning everything or just finetuning the last layer



results on CIFAR

Asymmetric Transfer

Self-Supervised Pretraining

When can't you use a pretrained network?

- there's no pretrained network for the kind of input data that you need
- or, none of the available ones work for you
- or, legally you are now allowed to use any of them

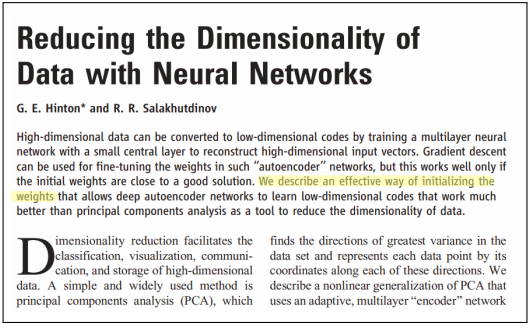
When can't you use a pretrained network?

- there's no pretrained network for the kind of input data that you need
- or, none of the available ones work for you
- or, legally you are now allowed to use any of them

What to do?

- if you have enough labeled data from a suitable source task
  - train the source network yourself
- you have unlabeled data of a suitable task, but no labeled data
  - consider unsupervised or self-supervised pretraining
- if you have no or very little data in general (labeled or unlabeled)
  - you're in trouble

This was one of the triggers that started the revival of neural network in 2006:



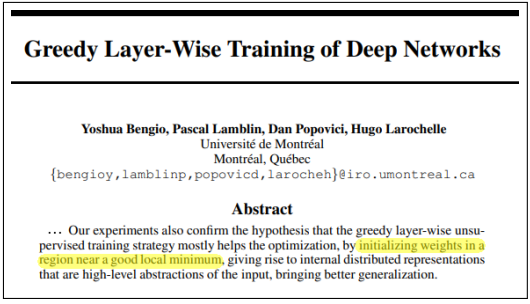
Reducing the Dimensionality of Data with Neural Networks

G. E. Hinton* and R. R. Salakhutdinov

High-dimensional data can be converted to low-dimensional codes by training a multilayer neural network with a small central layer to reconstruct high-dimensional input vectors. Gradient descent can be used for fine-tuning the weights in such "autoencoder" networks, but this works well only if the initial weights are close to a good solution. We describe an effective way of initializing the weights that allows deep autoencoder networks to learn low-dimensional codes that work much better than principal components analysis as a tool to reduce the dimensionality of data.

Dimensionality reduction facilitates the classification, visualization, communication, and storage of high-dimensional data. A simple and widely used method is principal components analysis (PCA), which finds the directions of greatest variance in the data set and represents each data point by its coordinates along each of these directions. We describe a nonlinear generalization of PCA that uses an adaptive, multilayer "encoder" network

Science 2006



Greedy Layer-Wise Training of Deep Networks

Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle
Université de Montréal
Montréal, Québec
{bengioy,lamblinp,popovicd,larocheh}@iro.umontreal.ca

Abstract

... Our experiments also confirm the hypothesis that the greedy layer-wise unsupervised training strategy mostly helps the optimization, by initializing weights in a region near a good local minimum, giving rise to internal distributed representations that are high-level abstractions of the input, bringing better generalization.

NIPS 2006

- pre-trained layers of the network one-by-one starting at the input
- learn weights by auto-encoding the signal that arrives from the previous layer

Abandoned soon after, not competitive by today's standards.

Continuous method for unsupervised representation learning:

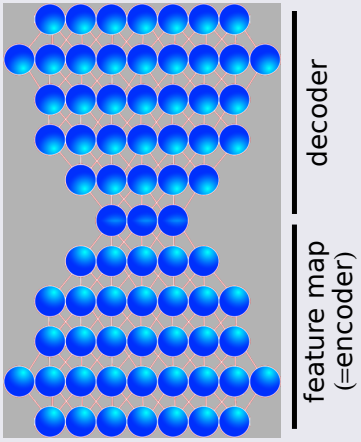
Autoencoder

Goal: Learn a feature map $\phi_\eta : \mathcal{X} \to \mathbb{R}^D$:

- Instead of a clasifier layer, attach a decoder network $g_{\eta'}$ to the feature map (essentially a mirrored copy of the feature layers)
- Learn parameters that allow best data construction

$$\min_{\eta,\eta'} \sum_{i=1}^{N} \|X_i - g_{\eta'}(\phi_\eta)(X_i)\|^2$$

- no labeled needed!



Probabilistic version (variational autoencoder) → lecture on generative models

Self-supervised pretraining

Observation: For supervised target tasks, one should use a supervised task for pretraining, even if it is not exactly the right one.

Idea: make up an auxiliary task for which the labels can be generated on-the-fly

Example: Computer Vision

- given: images $X_1, \ldots, X_N$

Auxiliary task:

- create auxiliary data, $(\tilde{X}_i, \tilde{Y}_i)_{i=1}^N$, as
  - auxiliary outputs, $\tilde{Y}_i$, are randomly choosen orientations $\{0°, 90°, 180°, 270°\}$
  - auxiliary images, $\tilde{X}_i$, are the original images, X_i, rotated by $\tilde{Y}_i$
- train a network on the auxiliary training dataset, $(\tilde{X}_i, \tilde{Y}_i)_{i=1}^N$,

Intuition: Features that work well for predicting if an image has been rotated or not need to contain information about scene geometry, object shape, appearance, etc.

Revisiting Self-Supervised Visual Representation Learning

Alexander Kolesnikov*, Xiaohua Zhai*, Lucas Beyer*
Google Brain Zürich, Switzerland

CVPR 2019

Family		ImageNet		
		Prev. top1	Ours top1	Ours top5
A	Rotation[11]	38.7	**55.4**	**77.9**
R	Exemplar[8]	31.5	46.0	68.8
R	Rel. Patch Loc.[8]	36.2	51.4	74.0
A	Jigsaw[34, 51]	34.7	44.6	68.0
R	Supervised RevNet50	74.8	74.4	91.9
R	Supervised ResNet50 v2	76.0	75.8	92.8
V	Supervised VGG19	72.7	75.0	92.3

Example: Natural Language Processing

- given: one or more long text documents

Auxiliary task:

- split text documents into sentences, $S_1, \ldots, S_T$
- create auxiliary data, $(\tilde{X}_i, \tilde{Y}_i)_{i=1}^N$, as
  - $\tilde{X}_i = (S_k, S_l)$ for $k, l \in \{1, \ldots, T\}$ are pairs of sentences
  - $\tilde{Y}_i = [\![l = k + 1]\!]$, specifies if the second sentences directly followed the first
- train a network on the auxiliary training dataset, $(\tilde{X}_i, \tilde{Y}_i)_{i=1}^N$,

Intuition: Features that work well for predicting if one sentence follows another must contain information about semantics (i.e. the meaning of words)

Examples: Natural Language Processing

Bert: Pre-training of deep bidirectional transformers for language understanding
J Devlin, MW Chang, K Lee, K Toutanova - arXiv preprint arXiv ..., 2018 - arxiv.org
We introduce a new language representation model called BERT, which stands for Bidirectional Encoder Representations from Transformers. Unlike recent language representation models, BERT is designed to pre-train deep bidirectional representations by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT representations can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language ...
☆ 99 Cited by 1456 Related articles All 7 versions »

BERT: `https://github.com/google-research/bert`:

- predicts vector representations of natural language words
- 340M parameters
- trained on Wikipedia (2.5 billion words) and BooksCorpus (800 million words)
- trained for 4 days on 64 TPUs
- state-of-the-art performance on a variety of language tasks

Examples: Natural Language Processing

Query:	What Is and Isn't Affected by the Government Shutdown?
result 1	Transportation Security Administration agents are among the most visible federal employees affected by the government shutdown.
result 2	Transportation Security Administration agents at Chicago Midway International Airport on Dec. 22, the first day of the government shutdown.
result 3	Damage from Hurricane Michael lingers throughout Marianna, Fla. The government shutdown has made things worse for many residents.
result 4	Scientists aboard a National Oceanic and Atmospheric Administration ship in San Juan, P.R. The administration has furloughed many workers because of the government shutdown.
result 5	Major federal agencies affected by the shutdown include the Departments of Agriculture, Commerce, Homeland Security, Housing and Urban Development, the Interior, Justice, State, Transportation, and the Treasury and the Environmental Protection Agency.

Take home message: unsupervised/self-supervised pretraining

(Variational) autoencoder are general-purpose method for feature learning.

For supervised tasks, self-supervised pretraining tends to work better.

For natural language processing, state-of-the-art representations (e.g. BERT) are based on self-supervised models.

Specific form of self-supervision depends on the task.

Part II

What to do if you cannot use pretrained features/finetuning at all:

- you have target labels, but they do not fit exactly the problem setting
  → weakly-supervised learning
  - e.g. in order to aim the water gun, one needs to know where exactly where the pigeon is located, but the annotation only states if there is a pigeon in the image or not.
- you have unlabeled data but no labeled data for the target task
  → unsupervised domain adaptation
  - e.g. I turn my pigeon detector into a business. People upload images of their terrace, and a pigeon detection model is automatically created for them.
- you have no data at all (not even unlabeled) for the target task
  → zero-shot learning
  - e.g. I expand my pigeon detection business to other animals: people send me the name of the animal they want to scare away, and a suitable model for this animal is automatically created for them.

Asymmetric Transfer

Domain Adaptation

Setting:

- one (or more) learning tasks with lots of labeled training data (sources)
  - $(X_1^{\text{src}}, Y_1^{\text{src}}), \ldots, (X_N^{\text{src}}, Y_N^{\text{src}})$
- one learning task with only unlabeled training data (target)
  - $X_1^{\text{tgt}}, \ldots, X_M^{\text{tgt}}$

The tasks typically have the same output classes but in different settings, e.g.

- classifying tumors cells into *benign* vs. *malignant* where the source is images of the *lung* and the target is the *thyroid*
- detecting pigeons in images, where the source is images from the Internet and the target is images from a webcam
- sentiment analysis for text, where the source is reviews from Amazon and the target is Donald Trump's tweets

Remember: we're interested only in the target task, the source task is just a tool.

Domain Adaptation

Most basic idea: train on the source task, use the trained model for the target task

Clearly, this can fail completely, e.g. if source and target are unrelated.

Even if source and target tasks fit together, it might or might not work:

Example

- source task: detect pigeons in images taken in summer
- target task: detect pigeons in images taken in winter

- if the model decides based on colors, the transfer will likely fail.
- if the model decides based on shape (contours), the transfer will likely work.

Insight: it matters what the source model has learned, in particular on the *features* is uses.

Reminder: training on source is what matters. Afterwards, nothing changes anymore.

Illustration: Features can be bad (top) or good (bottom)

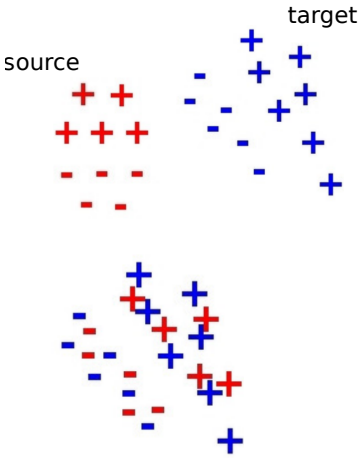
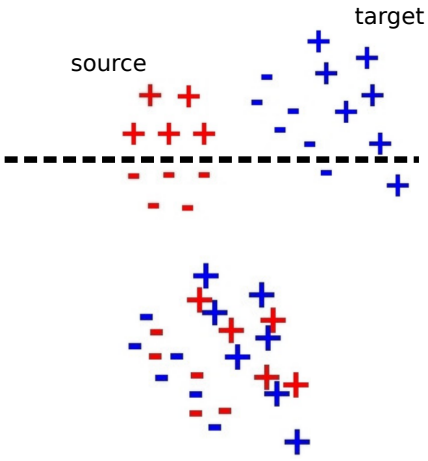


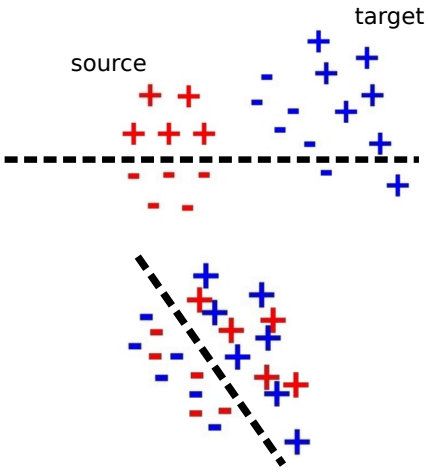
Illustration: Features can be bad (top) or good (bottom)



bad features: a classifer that works well on source might not work well on target

good features: any classifer that works well on source also works well on target

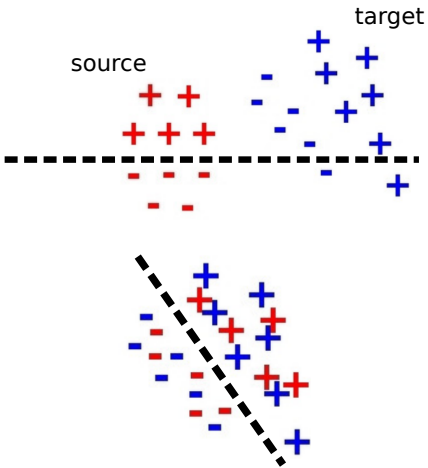
Illustration: Features can be bad (top) or good (bottom)



bad features: a classifer that works well on source might not work well on target

good features: any classifer that works well on source also works well on target

Illustration: Features can be bad (top) or good (bottom)



bad features: a classifer that works well on source might not work well on target

good features: any classifer that works well on source also works well on target

Idea:

- when training source model, encourage the network to extract features that will also work for the target task.

Technically:

- when looking at the feature representations of source and target data, they should be similar

How to measure similarity:

- compare elementary statistics, e.g. mean and covariance
  simple heuristic, but no good reason why it would work
- use a proper distance measure between data distributions
  - Kullback-Leibler divergence: hard to estimate from samples, can be infinite
  - total variation distance: hard to estimate from samples
  - Wasserstein distance (currently fashionable): hard to estimate from samples, few theoretic guarantees about resulting classifier
  - discrepancy distance: can be estimated and yields quality garantees!

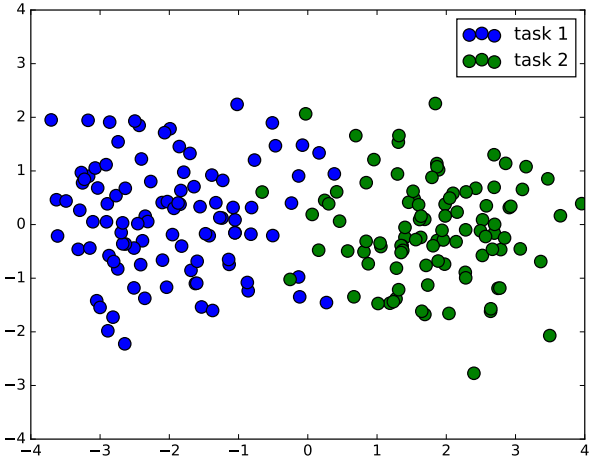
Discrepancy distance between unlabeled dataset:

- $S = \{X_1, X_2, \ldots, X_N\}, \quad S' = \{X'_1, X'_2, \ldots, X'_{N'}\}$
- $\mathcal{H}$: set of permitted classifiers, e.g. all linear ones

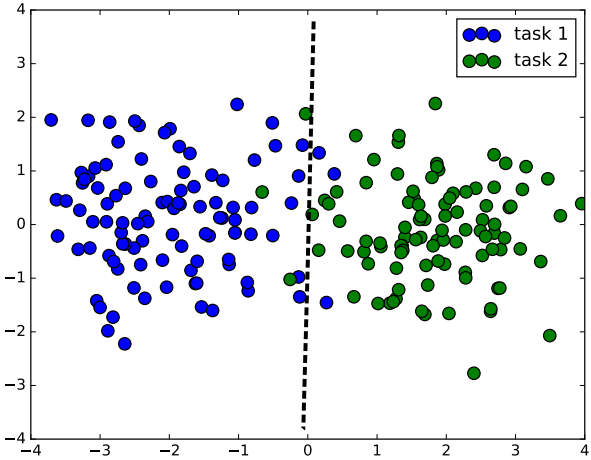
$$\text{disc}(S, S') = \max_{g,h \in \mathcal{H}} \left| \frac{1}{N} \sum_{i=1}^{N} [\![g(X_i) \neq h(X_i)]\!] - \frac{1}{N'} \sum_{i'=1}^{N'} [\![g(X'_{i'}) \neq h(X'_{i'})]\!] \right|$$

- For binary classification problems, equivalent to the accuracy of a classifier that tries to distinguish between S and S'.

Discrepancy distance: illustration

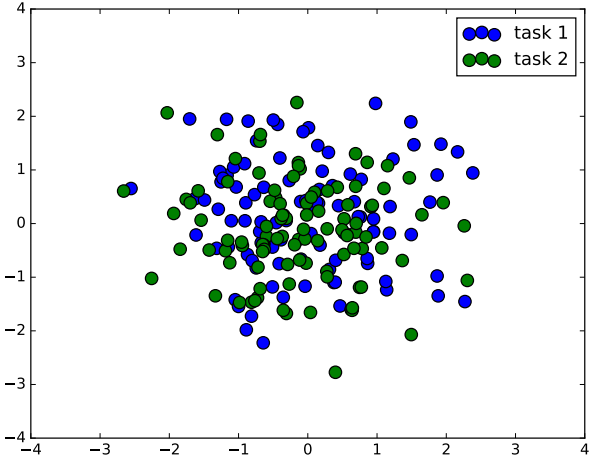


Discrepancy distance: illustration

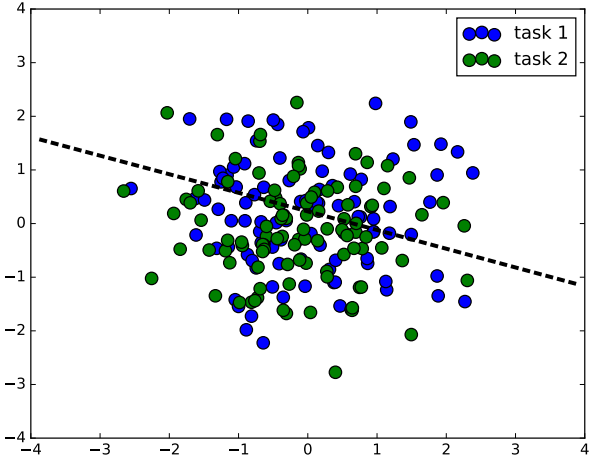


good separating linear classifier $\rightarrow$ large discrepancy (w.r.t. linear models)

Discrepancy distance: illustration



Discrepancy distance: illustration



no good separating linear classifier $\rightarrow$ small discrepancy (w.r.t. linear models)

Situation:

- data: $(X_1^{\text{src}}, Y_1^{\text{src}}), \ldots, (X_N^{\text{src}}, Y_N^{\text{src}}), \qquad X_1^{\text{tgt}}, \ldots, X_M^{\text{tgt}}$ (no labels for target)
- goal: learn a deep network $f_{\eta,\beta}$ with
  - η: parameters of feature map, $\phi_\eta : \mathcal{X} \to \mathbb{R}^F$
  - w: parameters of linear classification layer(s): $g_\beta : \mathbb{R}^F \to \mathcal{Y}$
- $\mathcal{H}$: set of binary linear classifiers: $\mathcal{H} \subset \{h : \mathbb{R}^F \to \{\pm 1\}\}$

Learning problem for domain adaptation: $\min_{\eta,\beta} \left[\mathcal{L}(\eta, \beta) + \text{disc}\left(\Phi_\eta^{\text{src}}, \Phi_\eta^{\text{tgt}}\right) \right]$

- $\mathcal{L}(\eta, \beta) = \sum_{i=1}^{N} \text{Loss}\big(Y_i^{\text{src}}, f_{\eta,\beta}(X_i^{\text{src}})\big)$
- $\text{disc}\left(\Phi_\eta^{\text{src}}, \Phi_\eta^{\text{tgt}}\right) = 1 - \min_{h \in \mathcal{H}} \left[\frac{1}{N} \sum_{i=1}^{N} \text{Loss}\big(+1, \phi_\eta(X_i^{\text{src}})\big) + \frac{1}{M} \sum_{i=1}^{M} \text{Loss}\big(-1, \phi_\eta(X_i^{\text{tgt}})\big) \right]$

Intuition: learn a classifier that

- 1) works well on the source
- 2) uses a feature representation in which source and target are indistinguishable

Not an easy optimization problem:

$$\min_{\eta,\beta} \Big[\mathcal{L}(\eta,\beta) + \text{disc}\left(\Phi_\eta^{\text{src}}, \Phi_\eta^{\text{tgt}}\right) \Big] = \min_{\eta,\beta} \Big[\mathcal{L}(\eta,\beta) + 1 - \min_{\tilde{\beta}} \tilde{\mathcal{L}}(\eta,\tilde{\beta}) \Big]$$
$$= \min_{\eta,\beta} \max_{\tilde{\beta}} \Big[\mathcal{L}(\eta,\beta) - \tilde{\mathcal{L}}(\eta,\tilde{\beta}) \Big] + 1$$

Features should:

- allow classification loss $\mathcal{L}$ to be minimized, but
- allow loss $\tilde{\mathcal{L}}$ from discrepancy to be maximized.

Not possible to compute SGD updates by single run of backpropagation.

Nice trick: *"gradient reversal layer"* [Ganin/Lempitsky: "Unsupervised Domain Adaptation by Backpropagation", ICML 2015]

Backpropagation with gradient reversal layer: with $\theta_f \equiv \eta$, $\theta_y \equiv \beta$, $\theta_d \equiv \tilde{\beta}$

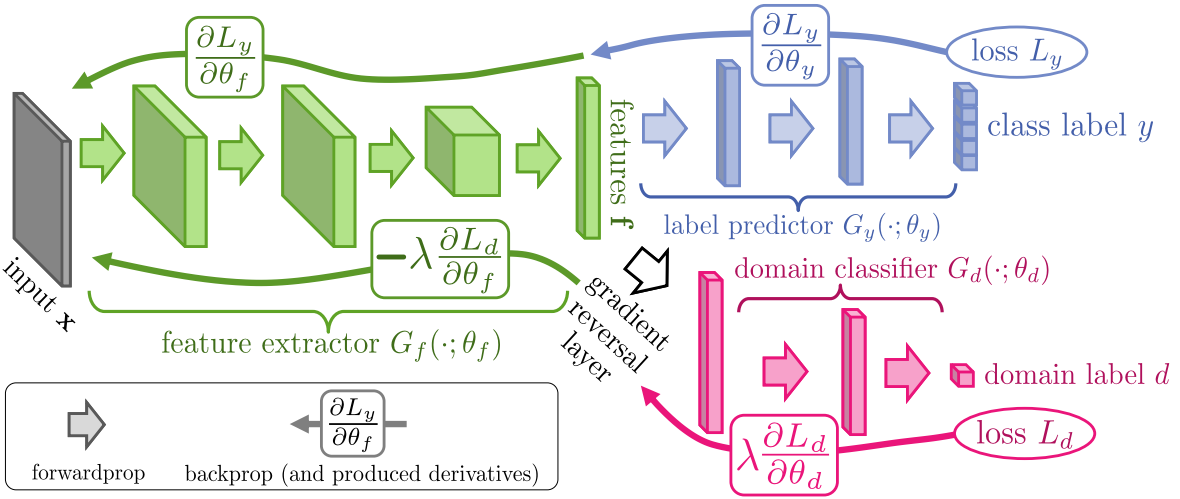


Image: [Ganin/Lempitsky: "Unsupervised Domain Adaptation by Backpropagation", ICML 2015]

When will it work?

Theorem [Ben-David *et al.*, 2010]

$$\text{error}_{\text{tgt}}(f) \leq \text{error}_{\text{src}}(f) + \text{disc}(\text{source}, \text{target}) + \lambda + \dots$$

where λ is the lowest possible error of any classifier on source+target together.

Example:

- target: pigeon detection in grayscale images
- source: pigeon detection in color images
- guess: λ probably small, method should work

Example:

- target: pigeon detection in images
- source: zebra detection in images
- guess: λ probably large, method should not work

Alternatives?

Domain mappings

- 1) learn mapping $F : \mathcal{X}^{\text{src}} \to \mathcal{X}^{\text{tgt}}$ from source data into target data
  - e.g., transform summer images to look like winter images,
  - e.g., transform liver cell images to look like thyroid cell images, etc.
- 2) convert source data $(X_i^{\text{src}})_{i=1}^N$ into data that looks like target data

$$\tilde{X}_i = F(X_i^{\text{src}}) \quad \text{for } i = 1, \ldots, N.$$

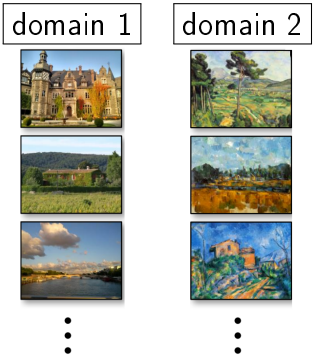
- 3) train a model, $f : \mathcal{X} \to \mathcal{Y}$ on pseudo-dataset $(\tilde{X}_1, Y_1), \ldots, (\tilde{X}_N, Y_N)$

Elegant idea, but converting data between domains is typically very hard.

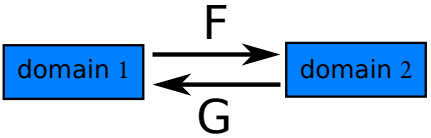
Example: Image-to-Image Translation with Cycle Consistency (CycleGAN)

Two datasets, unpaired:

- domain 1: $X_1, \ldots, X_N$
- domain 2: $X'_1, \ldots, X'_{N'}$



Idea: learn **two** conditional GANs:



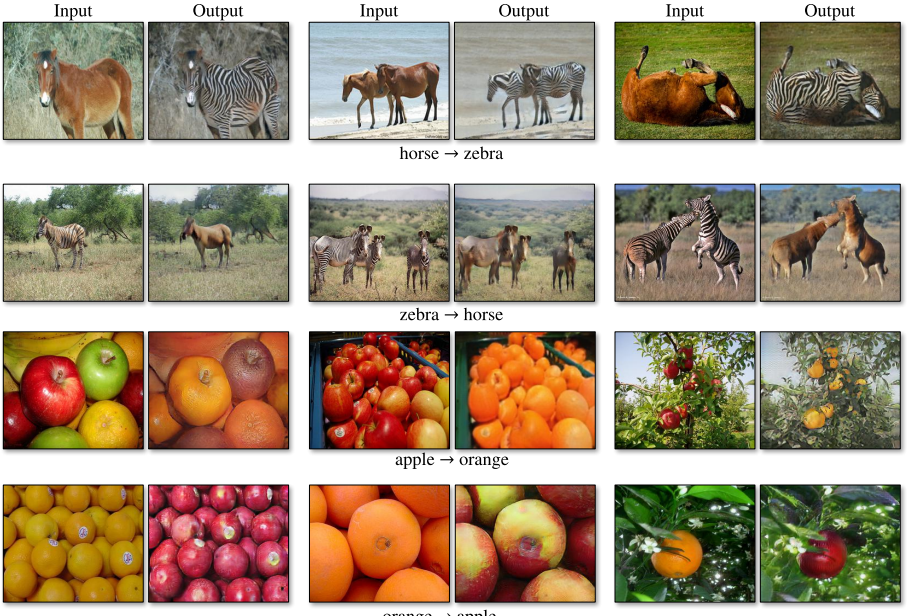
- F : domain1 $\rightarrow$ domain2
- G : domain2 $\rightarrow$ domain1

such that

- $F(G(X_i)) \approx X_i$ for $i = 1, \ldots, N$
- $G(F(X'_{i'})) \approx X'_i$ for $i' = 1, \ldots, N'$

[Zhu, Park, Isola, Efros. "Unpaired image-to-image translation using cycle-consistent adversarial networks". ICCV 2017]

Example: Image-to-Image Translation with Cycle Consistency (CycleGAN)



Images: [Zhu, Park, Isola, Efros. "Unpaired image-to-image translation using cycle-consistent adversarial networks". ICCV 2017]

Which part is the biggest problem of domain adaptation?

Which part is the biggest problem of domain adaptation?

Model Selection

How to chose hyperparameters, such as

- network architecture,
- regularization (if any),
- learning rate,
- number of epochs to train?

In ordinary learning, one uses cross-validation, or a validation set.

In domain adaptation, we have **no labeled data from the target set**, so we can't compute a validation error!

Model selection for domain adaptation

Model selection for domain adaptation

Idea 1) don't do model selection, just use some fixed parameters
$\rightarrow$ might work for writing a paper, certainly not for real-world tasks

Model selection for domain adaptation

Idea 1) don't do model selection, just use some fixed parameters
$\rightarrow$ might work for writing a paper, certainly not for real-world tasks

Idea 2) select parameters based on validation performance on the source
$\rightarrow$ usually fails: quality on source does not predict quality on target

Model selection for domain adaptation

Idea 1) don't do model selection, just use some fixed parameters
→ might work for writing a paper, certainly not for real-world tasks

Idea 2) select parameters based on validation performance on the source
→ usually fails: quality on source does not predict quality on target

Idea 3) annotate a bit of target data to use as validation set
→ why use domain adaptation then? why not use labels for *training*, too?

Model selection for domain adaptation

Idea 1) don't do model selection, just use some fixed parameters
→ might work for writing a paper, certainly not for real-world tasks

Idea 2) select parameters based on validation performance on the source
→ usually fails: quality on source does not predict quality on target

Idea 3) annotate a bit of target data to use as validation set
→ why use domain adaptation then? why not use labels for *training*, too?

Idea 4) **Reverse validation** [Zhong *et al.*, ECML 2010; Bruzzone *et al.*, TPAMI 2010]

a) use DA method to learn classifier for target using labeled source data
b) predict pseudo-labels for target data by applying the learned classifier
c) use DA method to learn classifier for source using pseudo-labeled target data
d) model select in c) by validation/cross-validation
e) check accuracy of learned classifier on source

Tedious to do, and not guaranteed to work well, but at least you are not cheating.

Take home message: unsupervised domain adaptation

Powerful concept for transfering information from a source domain with labels to a target domain with only unlabeled data.

Popular research topic, less popular for practical tasks.

Many theoretical results, but no guarantees that domain adaptation will work

See e.g. [Ben-David, Urner. "On the hardness of domain adaptation and the utility of unlabeled target samples". ALT 2012]

Proper model selection is tedious, most academic papers cheat.

Robust Learning from Untrusted Sources [Konstantikov, Lampert. ICML 2019]

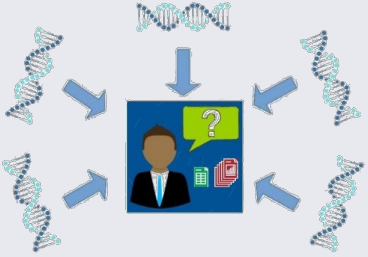
Situation:

- A small amount of data from a target task
- Large amounts of data from a source task, but unreliable (e.g. crowedsourced)

Question:

- Which sources to trust and how much?

Data from different labs



Solution:

- mathematical characterization of expected quality, depending on
  - discrepancy distance between target and sources
  - amounts of how much to rely on each source
- intuitive result: share from as many tasks as possible, but only from sufficiently similar ones

Part III

Asymmetric Transfer

Weakly-supervised Learning

Goal (target):

- learn a model: $f : \mathcal{X} \to \mathcal{Y}$
  - e.g. localize pigeons in images: $\mathcal{X} = \{\text{images}\}$, $\mathcal{Y} = \{\text{pigeon coordinates (if any)}\}$

Available (source):

- data: $(X_1, \tilde{Y}_1), \ldots, (X_N, \tilde{Y}_N) \subset \mathcal{X} \times \tilde{\mathcal{Y}}$, where labels in $\tilde{\mathcal{Y}}$ are *weaker* than in $\mathcal{Y}$,
- "weak" means: for any Y there is an obvious $\tilde{Y}$, whereas Y cannot be inferred from $\tilde{Y}$
  - e.g. $\mathcal{Y}$ is pigeon coordinates, $\tilde{\mathcal{Y}}$ is indicators if image shows a pigeon anywhere or not

Goal (target):

- learn a model: $f : \mathcal{X} \to \mathcal{Y}$
  - e.g. localize pigeons in images: $\mathcal{X} = \{\text{images}\}$, $\mathcal{Y} = \{\text{pigeon coordinates (if any)}\}$

Available (source):

- data: $(X_1, \tilde{Y}_1), \ldots, (X_N, \tilde{Y}_N) \subset \mathcal{X} \times \tilde{\mathcal{Y}}$, where labels in $\tilde{\mathcal{Y}}$ are *weaker* than in $\mathcal{Y}$,
- "weak" means: for any Y there is an obvious $\tilde{Y}$, whereas Y cannot be inferred from $\tilde{Y}$
  - e.g. $\mathcal{Y}$ is pigeon coordinates, $\tilde{\mathcal{Y}}$ is indicators if image shows a pigeon anywhere or not

regular annotation		weak annotation	
			
$Y = (0.4, 0.8, 0.1, 0.7)$	$Y = \emptyset$	$\tilde{Y} = \texttt{pigeon}$	$\tilde{Y} = \texttt{no pigeon}$

Images: Tanuj Handa (pigeon), Nadine Doerle (sparrow) from Pixabay

Why would there only be weak annotation? Often much faster to generate!

Weakly supervised learning methods are usually domain-dependent.
Method depends on *how* the full and the weak labels are related.

Example: Object Detection

$\mathcal{X} = \{\text{images}\}$, $\mathcal{Y} = \{\text{pigeon coordinates (if any)}\}$, $\tilde{\mathcal{Y}} = \{\texttt{pigeon}, \texttt{no pigeon}\}$

1) Use weak labels to train an image object classifier that outputs a *confidence score*, $g(X) = p(\texttt{pigeon}|X)$, for arbitrarily sized inputs X
2) Construct a object detector, i.e. $f : \mathcal{X} \to \mathcal{Y}$ as

$$f(X) = \operatorname*{argmax}_{R \subseteq X} f(R)$$

where $R \subseteq X$ ranges over all rectangular regions inside X.

Remaining problem: how to evaluate the argmax efficiently:

- branch-and-bound [Lampert *et al.* CVPR'08], object proposals e.g. [Uijlings *et al.* IJCV'13], [Ren *et al.*, NeurIPS'15]

Goal: learn a model: $f : \mathcal{X} \to \mathcal{Y}$

Available: weakly labeled data $(X_1, \tilde{Y}_1), \ldots, (X_N, \tilde{Y}_N) \subset \mathcal{X} \times \tilde{\mathcal{Y}}$

Extension: Self-Training

1) Use weakly labeled data to build an initial (weak) model, $f : \mathcal{X} \to \mathcal{Y}$
2) Apply f to the training set to create pseudo-labels $Y_i' = f(X_i)$, for $i = 1, \ldots, N$
3) Use some or all data with pseudo-labels to train (stronger) model, $f : \mathcal{X} \to \mathcal{Y}$
4) Potentially: repeat 2) and 3) multiple times

Problem: model selection

- if we have no fully annotated data, we cannot estimate how well f works.
- if we do have some fully annotated data, why not use it for training?

Take home message: weakly supervised learning

Many different variants, depending on what makes the supervision weak.

Solutions are usually task-specific: problem-specific network architecture, problem specific loss functions, etc.

Weak annotation sounds like a plausible idea for saving annotation effort but: results are typically weaker than with regular supervision, even taking labeling time into account.

Not obvious way how to do proper model selection.

Asymmetric Transfer

Strongly-supervised learning

Strongly-supervised learning

What about the opposite case to weakly-supervised: **strongly-supervised** learning

Goal (target):

- learn a model: $f : \mathcal{X} \to \mathcal{Y}$
  - e.g. localize pigeons in images: $\mathcal{X} = \{\text{images}\}$, $\mathcal{Y} = \{\text{pigeon coordinates (if any)}\}$

Available (source):

- data: $(X_1, \tilde{Y}_1), \ldots, (X_N, \tilde{Y}_N) \subset \mathcal{X} \times \tilde{\mathcal{Y}}$, where labels in $\tilde{\mathcal{Y}}$ are *stronger* than in $\mathcal{Y}$,
- "stronger" means: obtaining Y from $\tilde{Y}$ is easy, but $\tilde{Y}$ cannot be inferred from Y
  - e.g. $\mathcal{Y}$ is pigeon coordinates, $\tilde{\mathcal{Y}}$ is per-pixel annotation of pigeon vs. non-pigeon

Strongly-supervised learning

What about the opposite case to weakly-supervised: **strongly-supervised** learning

Goal (target):

- learn a model: $f : \mathcal{X} \to \mathcal{Y}$
  - e.g. localize pigeons in images: $\mathcal{X} = \{\text{images}\}$, $\mathcal{Y} = \{\text{pigeon coordinates (if any)}\}$

Available (source):

- data: $(X_1, \tilde{Y}_1), \ldots, (X_N, \tilde{Y}_N) \subset \mathcal{X} \times \tilde{\mathcal{Y}}$, where labels in $\tilde{\mathcal{Y}}$ are *stronger* than in $\mathcal{Y}$,
- "stronger" means: obtaining Y from $\tilde{Y}$ is easy, but $\tilde{Y}$ cannot be inferred from Y
  - e.g. $\mathcal{Y}$ is pigeon coordinates, $\tilde{\mathcal{Y}}$ is per-pixel annotation of pigeon vs. non-pigeon

One obvious solution: ignore the additional information:

1) create ordinary labels $(Y_i)_{i=1}^N$ from strong labels $(\tilde{Y}_i)_{i=1}^N$
2) train model on data $(X_1, Y_1), \ldots, (X_N, Y_N)$

Can we do better? That depends on the kind of additional information...

Example: model transfer

Available (source):

- a trained model: $g : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$ 'soft' outputs, e.g. $g(X) = p(Y|X)$
- dataset $(X_1, Y_1), \ldots, (X_N, Y_N)$ where $Y_i = g(X_i)$

Goal (target):

- learn a model: $f : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$ (or $f : \mathcal{X} \to \mathcal{Y}$)

Example: model transfer

Available (source):

- a trained model: $g : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$ 'soft' outputs, e.g. $g(X) = p(Y|X)$
- dataset $(X_1, Y_1), \ldots, (X_N, Y_N)$ where $Y_i = g(X_i)$

Goal (target):

- learn a model: $f : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$ (or $f : \mathcal{X} \to \mathcal{Y}$)

Obvious solution: just set $f \equiv g$.

But that might not be what we want, e.g.

- g is a big and inefficient model, we want f to be small and efficient.
- g might be the averaged outputs of many models, we want f to be a single network.

Knowledge Distillation

Given dataset $(X_1, g(X_1)), \ldots, (X_N, g(X_N))$ for a trained model: $g : \mathcal{X} \to \mathbb{R}^C$.
Find f_θ by solving

$$\min_\theta \sum_{i=1}^{N} \text{Loss}(\, g(X_i), f_\theta(X_i)\,)$$

for the cross-entropy loss between model outputs:

$$\text{Loss}(g(X), f(X)) = -\sum_{c=1}^{C} [g(X)^{1/\alpha}]_c \log [f(X)^{1/\alpha}]_c,$$

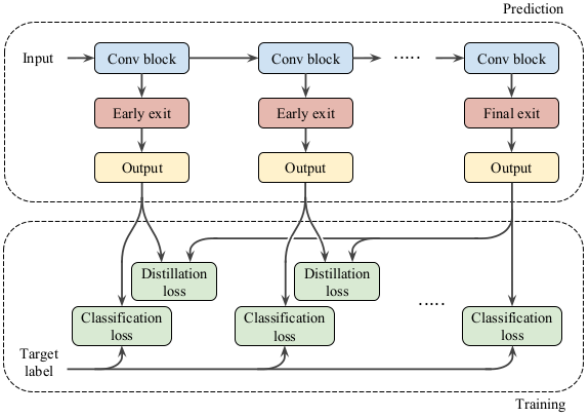
where $\alpha > 0$ is a temperature parameter.

Knowledge distillation often superior to learning from labels:

- faster convergence, e.g. [Phuong, Lampert. ICML 2019]
- resulting model has better prediction quality (if g is good).

Applications:

- Learning individual networks from ensembles of networks
- Learning small networks from large networks
- Learning any-time architecture [Phuong, Lampert. ICCV 2019]



Take home message: knowledge distillation

Powerful concept to transfer information from one model (architecture) to another.

Learning from another classifier's output often more effective than learning from labels.

Method of choice for some problems, e.g., for training very small and efficient models.

Allows using making use of unlabeled data (labels are created by the source model).

Asymmetric Transfer

Zero-shot learning

Zero-shot learning

Goal: learn a classifier $f : \mathcal{X} \to \mathcal{Y}$ without *any* training data for $\mathcal{Y}$ (labeled or unlabeled)

Zero-shot learning

Goal: learn a classifier $f : \mathcal{X} \to \mathcal{Y}$ without *any* training data for $\mathcal{Y}$ (labeled or unlabeled)

Clearly, this is impossible!

Zero-shot learning

Goal: learn a classifier $f : \mathcal{X} \to \mathcal{Y}$ without *any* training data for $\mathcal{Y}$ (labeled or unlabeled)

Clearly, this is impossible!

Well, not always... [Lampert *et al.* CVPR 2009], [Palatucci *et al.*, NeurIPS 2009]

Assume, you have training for other classes:

- $(X_1, Y'_1), \ldots, (X_N, Y'_N) \subset \mathcal{X} \times \mathcal{Y}'$ with $\mathcal{Y}' \cap \mathcal{Y} = \emptyset$

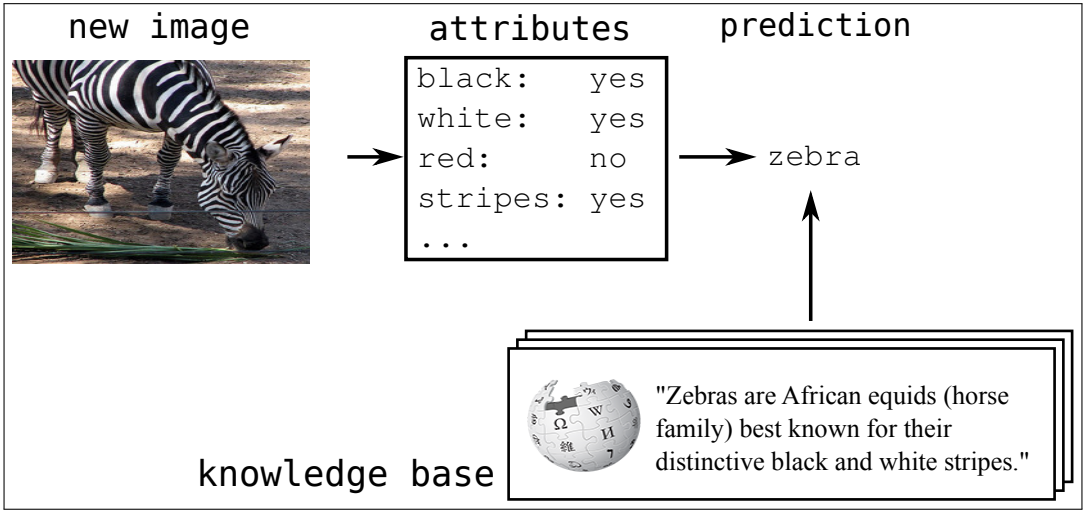
Assume, you have additional knowledge about all classes $\mathcal{Y} \cup \mathcal{Y}'$, e.g.

- a description in terms of some semantic *properties* (attributes), or
- the class names

Then, **zero-shot learning** is possible.

Example: attribute-based classification [Lampert *et al.*, CVPR 2009]

Idea: classifier $f : \mathcal{X} \to \mathcal{Y}$ based on presence or absence of attributes:



Example: attribute-based classification

Step 1) Learn classifiers for attributes **A**, **B**, **C** using other classes, $\mathcal{Y}'$:

pigeon	persian cat	polar bear	tiger
			
A=1, B=0, C=0	**A=1, B=1, C=0**	**A=0, B=1, C=0**	**A=0, B=1, C=1**

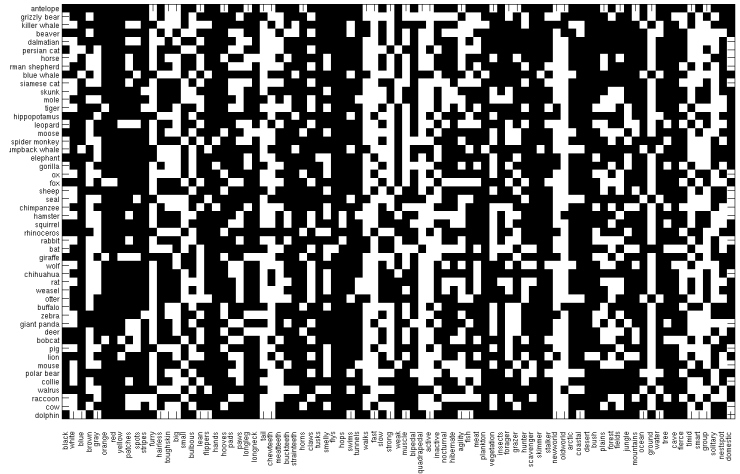
- from class labels we can also construct training sets for attributes

	+	−
A (e.g. black)	{persian cat} ∪ {pigeon} ∪ {tiger}	{polar bear}
B (e.g. white)	{persian cat} ∪ {polar bear} ∪ {tiger}	{pigeon}
C (e.g. has stripes)	{tiger}	{pigeon} ∪ {persian cat} ∪ {polar bear}

- train per-attribute classifiers $f_{\mathbf{A}}, f_{\mathbf{B}}, \ldots$
  - for a new image X, what's the probability of attributes **A**, **B**, and **C**?
    $f_{\mathbf{A}}(X) = p(\mathbf{A} = 1|X), \quad \ldots$

Example: attribute-based classification

Which class has which attributes? here: manually created



Could also be automatized, e.g. data mining on text documents.

Goal: learn classifier $f : \mathcal{X} \to \mathcal{Y}$

Step 2) for any image X to be classified

- use trained attribute classifiers to predict attributes vector

$$a_X = (f_{\mathbf{A}}(X), f_{\mathbf{B}}(X), \cdots) \in \mathbb{R}^{\#\text{attributes}}$$

- for any class $Y \in \mathcal{Y}$, look up attribute vector $a_Y \in \{0, 1\}^{\#\text{attributes}}$
- define zero-shot classifier $f : \mathcal{X} \to \mathcal{Y}$ as

$$f(X) = \operatorname*{argmin}_{Y \in \mathcal{Y}} d(a_X, a_Y)$$

where d is a geometric or probabilistic distance measure, e.g. Euclidean

Example: embedding-based zero-shot learning

Goal: learn a classifier $f : \mathcal{X} \to \mathcal{Y}$

Given:

- $S = \{(X_1, Y'_1), \ldots, (X_N, Y'_N)\} \subset \mathcal{X} \times \mathcal{Y}'$ with $\mathcal{Y}' \cap \mathcal{Y} = \emptyset$

Method:

- compute
  - feature vectors $\phi(X_i) \in \mathbb{R}^D$ for $i = 1, \ldots, N$ (e.g. pretrained network)
  - vector representation, $\psi(Y) \in \mathbb{R}^{D'}$, of class names for all classes $\mathcal{Y} \cup \mathcal{Y}'$ (e.g. BERT)
- use available training data, S, to learn compatibility function

$$F : \mathbb{R}^D \times \mathbb{R}^{D'} \to \mathbb{R}, \qquad \text{e.g. bilinear: } F(v, w) = v^\top M w \text{ for } M \in \mathbb{R}^{D \times D'}$$

- construct zero-shot classifier $f : \mathcal{X} \to \mathcal{Y}$ as

$$f(X) = \operatorname*{argmax}_{Y \in \mathcal{Y}} F(\phi(X), \psi(Y))$$

Variant: Domain Generalization

Given: many learning tasks with data

1) e.g. object detection for many object categories
2) e.g. movie recommendations for many users
3) e.g. tumor cell classification for many hospitals

Goal: create a model for a new task for which no training data is available

1) e.g. object detection for a new object category
2) e.g. movie recommendations for a new user
3) e.g. tumor cell classification for a new hospital

Variant: Domain Generalization

Given: many learning tasks with data

1) e.g. object detection for many object categories
2) e.g. movie recommendations for many users
3) e.g. tumor cell classification for many hospitals

Goal: create a model for a new task for which no training data is available

1) e.g. object detection for a new object category
2) e.g. movie recommendations for a new user
3) e.g. tumor cell classification for a new hospital

Leverage ideas from zero-shot learning and domain adaptation.

Variant: Domain Generalization

Given: many learning tasks with data

1) e.g. object detection for many object categories
2) e.g. movie recommendations for many users
3) e.g. tumor cell classification for many hospitals

Goal: create a model for a new task for which no training data is available

1) e.g. object detection for a new object category
2) e.g. movie recommendations for a new user
3) e.g. tumor cell classification for a new hospital

Leverage ideas from zero-shot learning and domain adaptation.

Assume that some contextual information about each task is available:

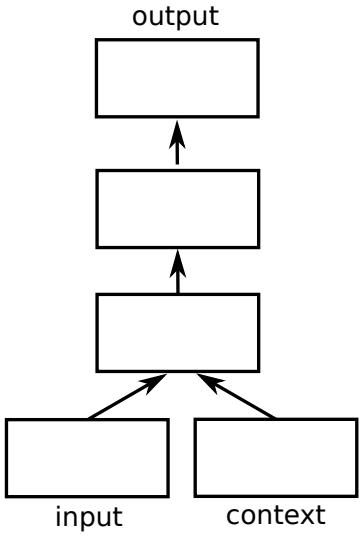
1) e.g. object category name
2) e.g. user's browsing history
3) e.g. hospital location, owner, public statistics, ...

Given:

- T tasks with, for $t = 1, \ldots, T$:
  - training data, $S'_t = ((X_1^{(t)}, Y_1^{(t)}), \ldots, (X_{N_t}^{(t)}, Y_{N_t}^{(t)})$
  - task context vector, μ_t

Method:

- build a network with two inputs:
  - X: ordinary model input
  - μ: context vector
- train network f with data of all source tasks:
  - $S' = S'_1 \cup S'_2 \cup \cdots \cup S'_T$ with $S'_t = ((X_1^{(t)}, \mu_t, Y_1^{(t)}), \ldots, (X_{N_t}^{(t)}, \mu_t, Y_{N_t}^{(t)})$
- for new example X of a task with context μ_{tgt}: model prediction is $f(X, \mu)$



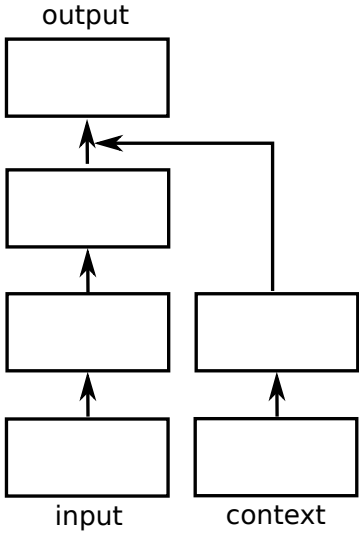
Given:

- T tasks with, for $t = 1, \ldots, T$:
  - training data $S'_t = ((X_1^{(t)}, Y_1^{(t)}), \ldots, (X_{N_t}^{(t)}, Y_{N_t}^{(t)})$
  - task context vector μ_t

Method:

- build two networks
  - one model extracts features, $\phi_\eta : \mathcal{X} \to \mathcal{Y}$
  - one model predicts a matrix of network weights, $\beta : \{\text{context}\} \to \{\text{weight matrices}\}$
  - use output of β as weights for classifier
- train network jointly with data of all source tasks
- for any context, μ, model prediction is

$$f(X) = \phi_\eta(X)^\top \beta(\mu)$$



Take home message: zero-shot learning

Extreme form of transfer: all information from context, none from target data.

Model selection is tedious (simulate zero-shot setting on available data).

To be honest, results are rather weak. Similar quality as with a few training examples.
$\rightarrow$ can be "last resort": we really need a classifier, there really is no data.

Variants of zero-shot are actually used in practice, e.g. to make recommendations for newly registered users.

Part IV

Symmetric Transfer

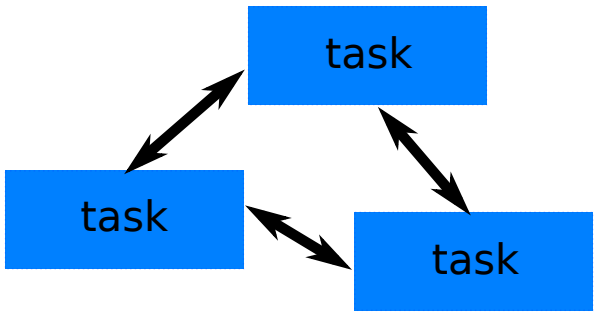
Symmetric Transfer

Multi-task Learning

General idea: we have multiple tasks that we want to solve. We have some training data for each of them, but hope that by sharing information or data between the tasks, results could get even better.

Situations/Methods:

- **multi-task learning**
  and variants...



Examples:

- personalized speech recognition (each speaker is a tasks)
- benign/malignant assessment of tumor cells (each cancer type is a task)
- all of computer vision: each of image compression, tracking, image classification, object detection, semantic segmentation, optical flow estimation, estimating 3D scene geometry, . . . , is a task.
- all of natural language processing: each of sentiment analysis, semantic parsing, translation, question answering, extailment, summarization, . . . is a task.
- learning to play computer games (each game is a task)
- natural language translation (each language is a task)
- international handwriting recognition (each language or each script is a task)

Given: multiple datasets, $S_1, \ldots, S_T$

- $S^{(t)} = (X_1^{(t)}, Y_1^{(t)}), \ldots, (X_{N_t}^{(t)}, Y_{N_t}^{(t)}) \subset \mathcal{X}^{(t)} \times \mathcal{Y}^{(t)}$, for $t = 1, \ldots, T$
- different tasks can have
  - different input spaces,
  - different output spaces,
  - different number of examples.

Goal: learn models for all tasks, $f^{(t)} : \mathcal{X}^{(t)} \to \mathcal{Y}^{(t)}$ for $t = 1, \ldots, T$

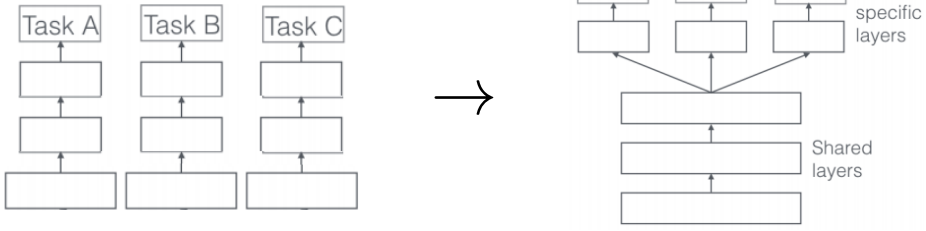
Trivial solution:

- learn each model, $f^{(t)}$, using the data $S^{(t)}$ for $t = 1, \ldots, T$.

Can we do better? That depends on how the tasks are related...

Situation 1: all tasks have the same input space, $\mathcal{X}^{(1)} = \mathcal{X}^{(2)} = \cdots = \mathcal{X}^{(T)} = \mathcal{X}$

Common idea: find a data representation (shared feature map) that works well for all tasks. Based on these features learn individual per-task classifiers.



Properties:

- If such features exist, we can learn them from all data of all tasks
  $\rightarrow$ we can use a bigger network than when learning for a single task
- If such features do not exists, trying to rely on them anyway will make quality of model worse than learning tasks separately $\rightarrow$ negative transfer

Parameterize multi-task network:

- η: parameters of shared part
- $\beta_1, \ldots, \beta_T$: parameters of individual parts

Objective function: each task has its own training loss

$$\mathcal{L}^{(t)}(\eta, \beta_t) = \frac{1}{N_t} \sum\nolimits_{i=1}^{N_t} \text{Loss}\big(Y_i^{(t)}, f_{\eta,\beta_t}^{(t)}(X_i^{(t)}) \big)$$

For numeric optimization, combine by summing or averaging them:

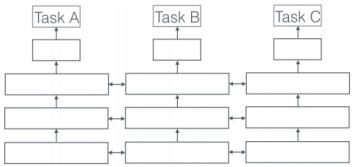
$$\min_{\eta,\beta_1,\ldots,\beta_T} \mathcal{L}^{\text{MT}}(\eta, \beta_1, \ldots, \beta_T) \qquad \text{for} \qquad \mathcal{L}^{\text{MT}}(\eta, \beta_1, \ldots, \beta_T) = \frac{1}{T} \sum_{t=1}^{T} \mathcal{L}^{(t)}(\eta, \beta_t)$$

Caveat: we might sacrifice quality on one task for sufficient gain on some other one.

Variant: why would exactly the same features be best for all tasks? Let's learn individual ones, but add a term to the objective that encourages they are all not too different from each other.

Parametrize:

- $\eta_1, \ldots, \eta_T$: parameters of individual feature maps
- $\beta_1, \ldots, \beta_T$: parameters of individual classifiers



Optimization:

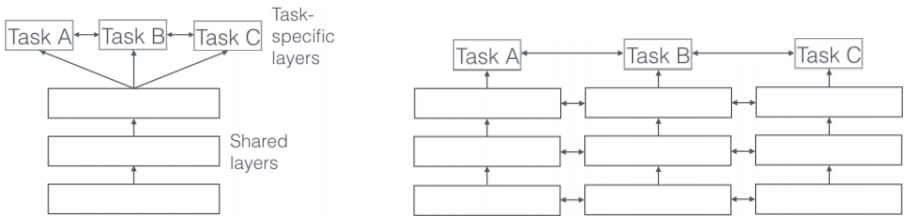
$$\min_{\eta_1,\ldots,\eta_T,\beta_1,\ldots,\beta_T} \frac{1}{T}\sum_{t=1}^{T} \mathcal{L}^{(t)}(\eta_t, \beta_t) + \lambda \sum_{s \neq t} \|\eta_s - \eta_t\|^2$$

Properties:

- Allows smooth transition between
  - no sharing: $\lambda = 0$
  - fully shared features: $\lambda \to \infty$

Situation 2: all tasks also have the same output space, $\mathcal{Y}^{(1)} = \mathcal{Y}^{(2)} = \cdots = \mathcal{Y}^{(T)} = \mathcal{Y}$

Idea: also encourage the classifier parts to be similar to each other



Optimization:
$$\min_{\eta_1,\ldots,\eta_T,\beta_1,\ldots,\beta_T} \frac{1}{T}\sum_{t=1}^{T} \mathcal{L}^{(t)}(\eta_t,\beta_t) + \lambda \sum_{s\neq t} \|\eta_s - \eta_t\|^2 + \mu \sum_{s\neq t} \|\beta_s - \beta_t\|^2$$

Properties:

- equivalent to learning shared reference parameters η, β and treating each classifier as a modification of it: $\eta_t = \eta + \delta_t$ and $\beta_t = \beta + \epsilon_t$
- can be expected to work well if all tasks are only minor modifications of each other

Situation 3: dataset for all tasks consists of the same input data

- $S^{(t)} = (X_1, Y_1^{(t)}), \ldots, (X_N, Y_N^{(t)}) \subset \mathcal{X} \times \mathcal{Y}^{(t)}, \quad$ for $t = 1, \ldots, T$

Example: classification of multiple objects in images

- $\mathcal{X} = \{\text{images}\}$
- $\mathcal{Y}_1 = \{\pm 1\}$: "is there a cat in the image or not"?
- $\mathcal{Y}_2 = \{\pm 1\}$: "is there a dog in the image or not"?
- etc.

We can apply the same multi-task learning techniques as before, but:

- we do not have more data to learn features than when learning tasks separately
  $\rightarrow$ at best, we benefit by sharing information between classifiers

Prediction of cardiovascular (CV) risk factors from retinal fundus photographs

- 48 layers ConvNet
- pretrained on ImageNet
- fine-tuned on 284.335 retinal images (TS of 13,000 patients)

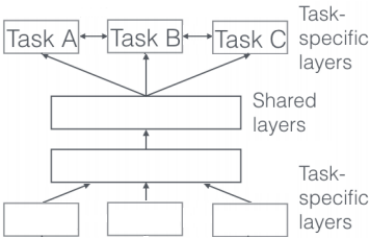
For each image, output multiple CV risk factors:

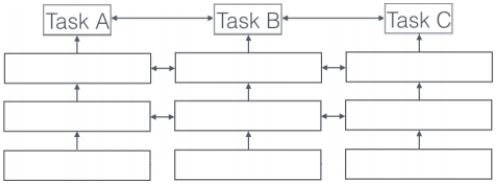
- task 1: predict age (average error: 3.3 years)
- task 2: predict gender (AUC 97%)
- task 3: predict smoking status (AUC 71%)
- task 4: HbA1c level (within 1.39%),
- task 5: systolic blood pressure (within 11.23mmHg)
- task 6: predict cardiovascular disease risk (AUC=70%)

(Poplin et al., Nature 2018)

Situation 4: tasks might have different input spaces (but are nevertheless related)

Idea: learn individual features, but share intermediate layers





Properties:

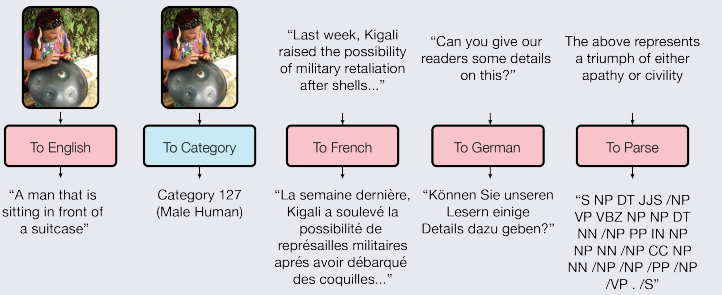
- lots of design choices: how many layers separately, how many shared, etc.

Can we have one model that solves all tasks?

MultiModel by Google

Train one model that solves 8 very heterogeneous tasks on different input modalities:

- detect objects, provide captions, recognize speech, translate between four pairs of language, grammatical parsing

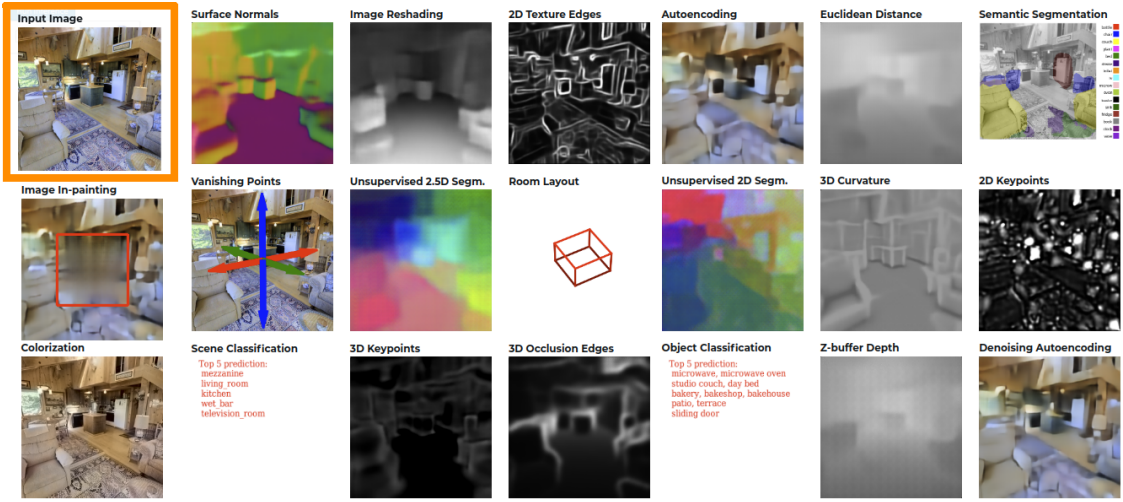


(Kaiser et al., 2017)

Caveat: it's only a proof-of-concept, not state-of-the-art on any tasks.

Example: Multi-task/multi-modal/Multi-label learning

Taskonomy `http://taskonomy.stanford.edu/` [Zamir *et al.*, CVPR 2018, best paper award]



A single model trained to solve 21 computer vision tasks.

Take home message: multi-task learning

If multiple tasks are to be learned, multi-task learning can often improve model quality.

Improvement comes from ability to use a stronger model for learning (shared) features.

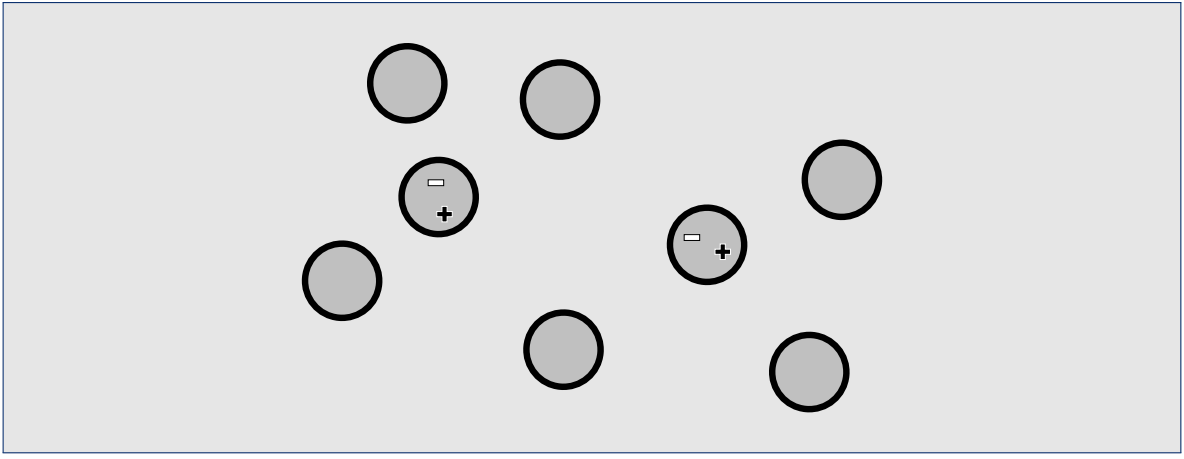
Negative transfer is possible, e.g. if assumptions on how tasks are related to not hold.

Other variants: multi-label learning, multi-target learning, multi-modal learning.

Sometimes used not to get better prediction quality of each task, but to get a compact and/or more convenient model of all tasks.

Active research: multi-task learning when some tasks have only unlabeled data

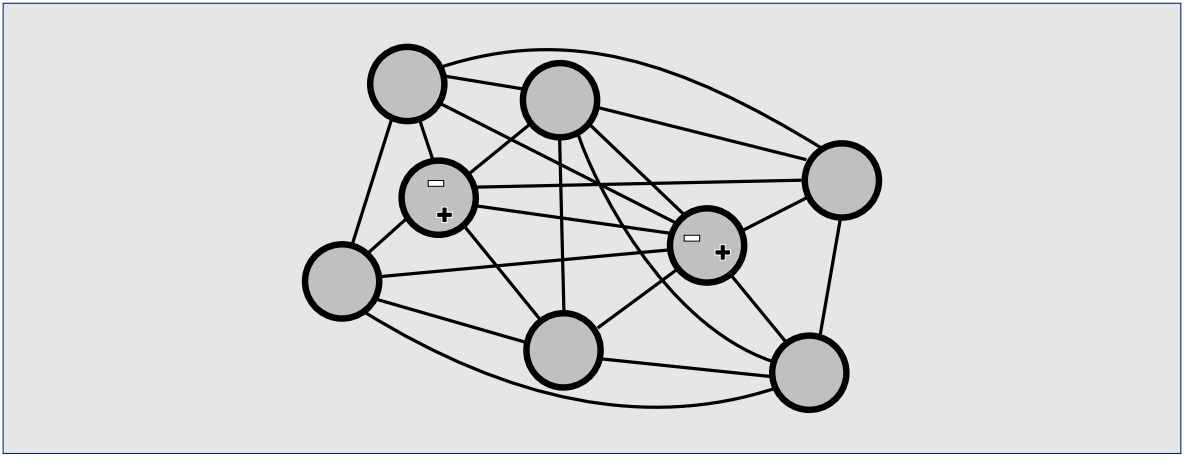
What if some of the tasks only have unlabeled data? [Pentina, Lampert. ICML 2017]



Given: tasks (circles), most of which have only unlabeled data

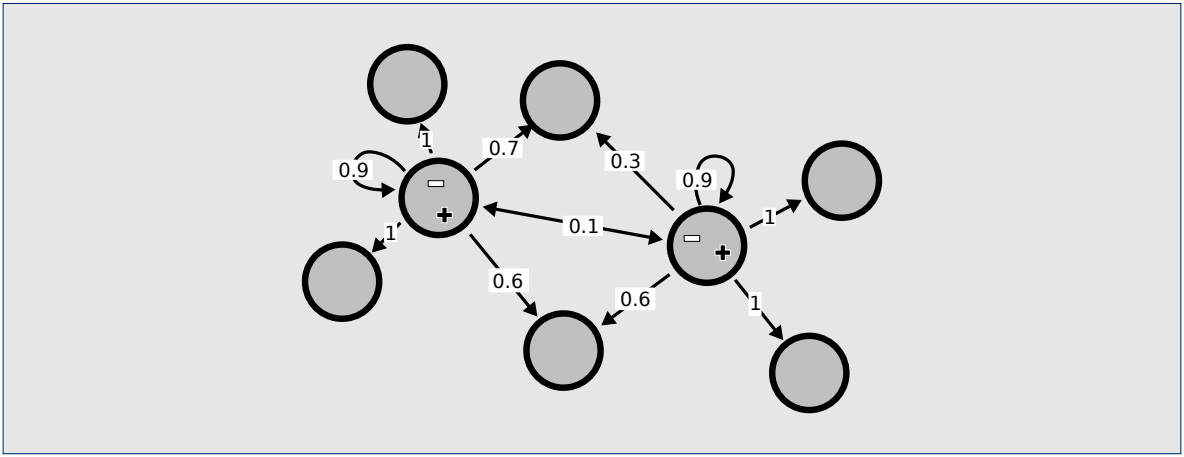
Active research: multi-task learning when some tasks have only unlabeled data

What if some of the tasks only have unlabeled data? [Pentina, Lampert. ICML 2017]



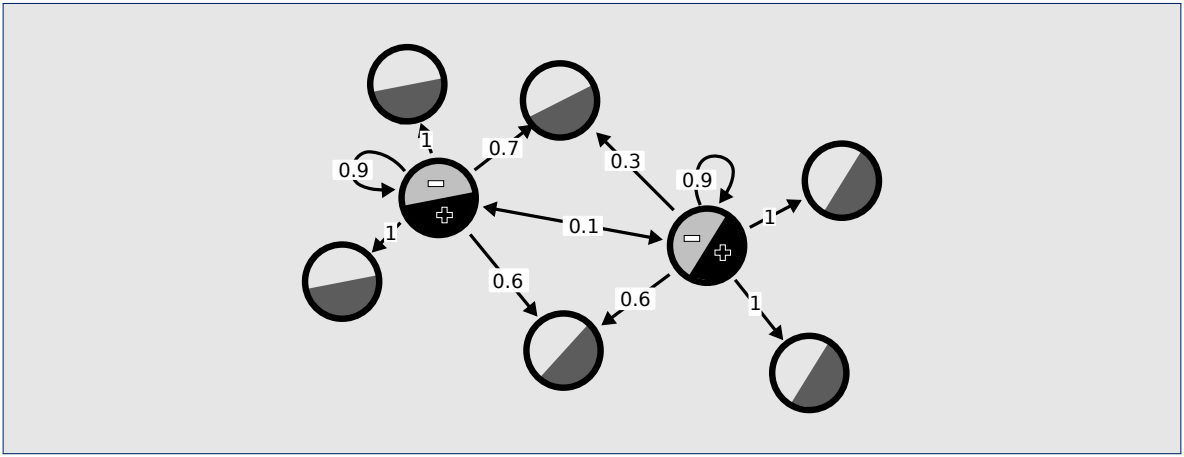
Step 1: compute pairwise discrepancies ("how well can a classifier tell the datasets apart")

What if some of the tasks only have unlabeled data? [Pentina, Lampert. ICML 2017]



Step 2: assign task-to-task weights based on task similarity

What if some of the tasks only have unlabeled data? [Pentina, Lampert. ICML 2017]



Step 3: learn classifier for each task by multi-source domain adaptation

Conclusion

Take home messages: Transfer Learning

Transfer learning allows using data/information/models from other learning task.
$\rightarrow$ we can learn models for situations where data would otherwise be insufficient

Some situations/methods have high practical relevance:

- pretrained features, finetuning, knowledge distillation, multi-task learning

Others situations/methods one should rather try to avoid in practice (if possible):

- domain adaptation (try collecting target labels)
- weakly-supervised learning (try collecting proper labels)
- zero-shot learning (try collecting target data)

Negative transfer is always possible. Make sure you use proper model selection!