Deep Learning (BEV033DLE)
Lecture 7 Training Neural Networks 0.

Czech Technical University in Prague

- Application goals
- Model selection, project pipeline
- Data collection, training/validation/test set
- Diagnosis, bottlenecks, debugging
- Overfitting, regularisation, early stopping
- Adjusting hyperparameters

Application goals

- Fix the project goals and the error metric you are going to use for it.
- Determine a reasonable performance level you expect to achieve.
- Remember that the performance metric usually differs from the objective function (loss function) used for the learning task.

Example 1. (Classification with rejection)
In some classification applications we allow rejections. Historical examples are

- Zip code recognition (this is where MNIST comes from)
- Google Street view: detect and transcribe house numbers in images (this is where SVHN comes from)

(1) If we can assign a cost to rejection: Extend the decision space of the predictor by an extra state "r" and assign a loss value ε to it.

$$h\colon \mathcal{X} \to \mathcal{D} = \mathcal{Y} \cup \{\text{“r”}\} \quad \ell(y,d) = \begin{cases} 0 & \text{if } d = y \\ \varepsilon & \text{if } d = \text{“r”}, \\ 1 & \text{otherwise} \end{cases}$$

Application goals

Example 1. (cont'd)
The optimal decision

$$h(x) = \arg\min_{d \in \mathcal{D}} \sum_{y \in \mathcal{Y}} p(y \mid x) \ell(y, d)$$

becomes

$$h(x) = \begin{cases} \arg\max_{y \in \mathcal{Y}} p(y \mid x) & \text{if } 1 - \max_{y \in \mathcal{Y}} p(y \mid x) < \varepsilon \\ \text{"r"} & \text{if } 1 - \max_{y \in \mathcal{Y}} p(y \mid x) \geqslant \varepsilon \end{cases}$$

(2) Alternatively, we might want to minimise the error rate constraining at the same time the rejection rate or, vice versa.

Simple practical approach:

- Learn the classifier using the log-likelihood of the training data as objective
- When applying the predictor, use the optimal decision with rejection as given above.

If for instance, one of the project goals is to keep the rejection frequency below some threshold, tune the value of ε using the validation set (see below).

Baseline model

Example 2. (incommensurable losses)
Consider large scale medical screening applications with small a-priory probability of the positive outcome ($k = 1$, disease). A false positive event incurs the cost of an unnecessary closer examination, whereas a false negative event might incur a fatality in the worst case.

We might want to minimise the frequency of false positive events constraining at the same time the frequency of false negative events. Equivalently: maximise the precision constraining the recall (from below).

It is known that the optimal decision rule for this task is

$$h(x) = \begin{cases} 1 & \text{if } \frac{p(x \mid k=1)}{p(x \mid k=0)} > \tau \\ 0 & \text{otherwise} \end{cases}$$

where τ is some threshold.

Simple practical approach:

- Learn the classifier using the log-likelihood of the training data as objective
- Use the predictor

$$h(x) = \begin{cases} 1 & \text{if } \frac{p(k=1 \mid x)}{p(k=0 \mid x)} > \tau' \\ 0 & \text{otherwise} \end{cases}$$

and tune τ' on the validation set.

Baseline model

Establish a baseline end-to-end system:

- Choose an appropriate model architecture, preferably up to only a few structural parameters (e.g. number of neurons in some of the hidden layers). This includes the choice of activation functions for the individual layers.
- Choose the criterion for the learning task, e.g. conditional log-likelihood if you learn a classifier.
- Choose the optimiser, e.g. SGD with momentum as a rule, possibly with decaying learning rate.
- Choose a set of regularisers you will try to use for preventing overfitting, i.e. bounding the generalisation error. They can range from weight decay and early stopping to dropout and batch normalisation.
- Is it possible to utilise transfer learning? I.e. learn the lower level features (first layers of the network) from a different task with more available training data.

Pipeline: Set up a train-test environment with a working cycle for improvement steps. This includes infrastructure for monitoring results, high level debugging, saving intermediate results etc.

Data: training/validation/test sets

You will almost certainly have hyperparameters in your pipeline: architecture parameters, optimiser parameters (e.g. learning rate and momentum), regulariser parameters (e.g. dropout rate). Therefore you should split the available data into three sets:

- training set: used for training the predictors,
- validation set: used for choosing the best predictor among those learned with different hyperparameter settings,
- test set: used for evaluating the finally chosen predictor.

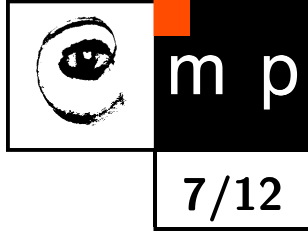
Use the Hoeffding inequality for choosing the proper test set size m

$$\mathbb{P}\Big(|R(h) - R_{\mathcal{T}^m}(h)| > \varepsilon\Big) < 2e^{-\frac{2m\varepsilon^2}{(\triangle\ell)^2}}.$$

Similarly, when choosing the best predictor from a finite set $\mathcal{H}$ of predictors learned with different hyperparameter settings, the proper size m of the validation set can be estimated from

$$\mathbb{P}\Big(\max_{h\in\mathcal{H}}|R(h) - R_{\mathcal{T}^m}(h)| > \varepsilon\Big) < 2|\mathcal{H}|e^{-\frac{2m\varepsilon^2}{(\triangle\ell)^2}}.$$

Data: training/validation/test sets



Notice, that in both cases $R(h)$ means the expected loss w.r.t. the chosen error metric ℓ.

The validation/test set sizes obtained from these inequalities are sufficient upper bounds. They are usually hard to meet.

First pipeline checks/Debugging

(1) Check the behaviour of the training loss and training error. If it is unsatisfactory:

- do low level debugging. If you have your own version of backprop, check behaviour of the gradient (see 2. lab). Try learning on a tiny data set or, if possible, on artificial data.
- is the model too simple?
- is it necessary to improve the learning algorithm? E.g. by better tuning the learning rate.
- is the quality of the data sufficient? E.g. noise in inputs, noisy labels.

(2) If the performance on the training set is acceptable, measure the performance on the test set. If it is much worse than the training set performance:

- do you need more training data? Is it possible to use data augmentation?
- experiment with training set sizes (on logarithmic scale),
- try to reduce the generalisation error by improving regularisation (e.g. using dropout or batch normalisation)

Improving the pipeline

- Monitor training/test losses/errors during learning (see lecture 5.)
- Identify bottlenecks and visualise the model's worst errors.
- repeatedly try incremental changes such as adjusting hyperparameters, based on findings.

Early stopping

- Monitor the exponentially weighted average validation error during training and stop it when the validation error starts to increase.
- This is an empirical approach. Theoretical guarantees exist so far only for kernel based regression learning. They are based on the following bias-variance trade off

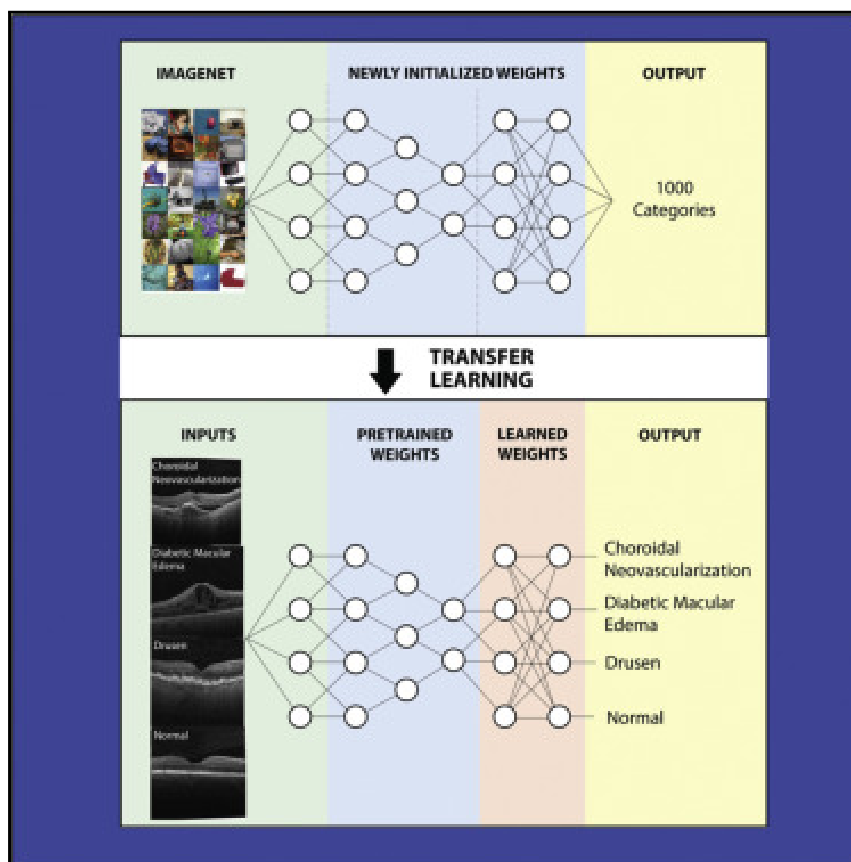
$$R(h_m^{(t)}) - R(h_{\mathcal{H}}) = \big[R(h_m^{(t)}) - R(h^{(t)})\big] + \big[R(h^{(t)}) - R(h_{\mathcal{H}})\big],$$

where $h_m^{(t)}$ denotes the predictor trained on $\mathcal{T}^m$ after t training epochs and $h^{(t)}$ denotes the predictor trained on the true risk after t training epochs. The two expressions in the r.h.s. can be upper bounded in terms of m, t and the learning rate.

Improving the pipeline

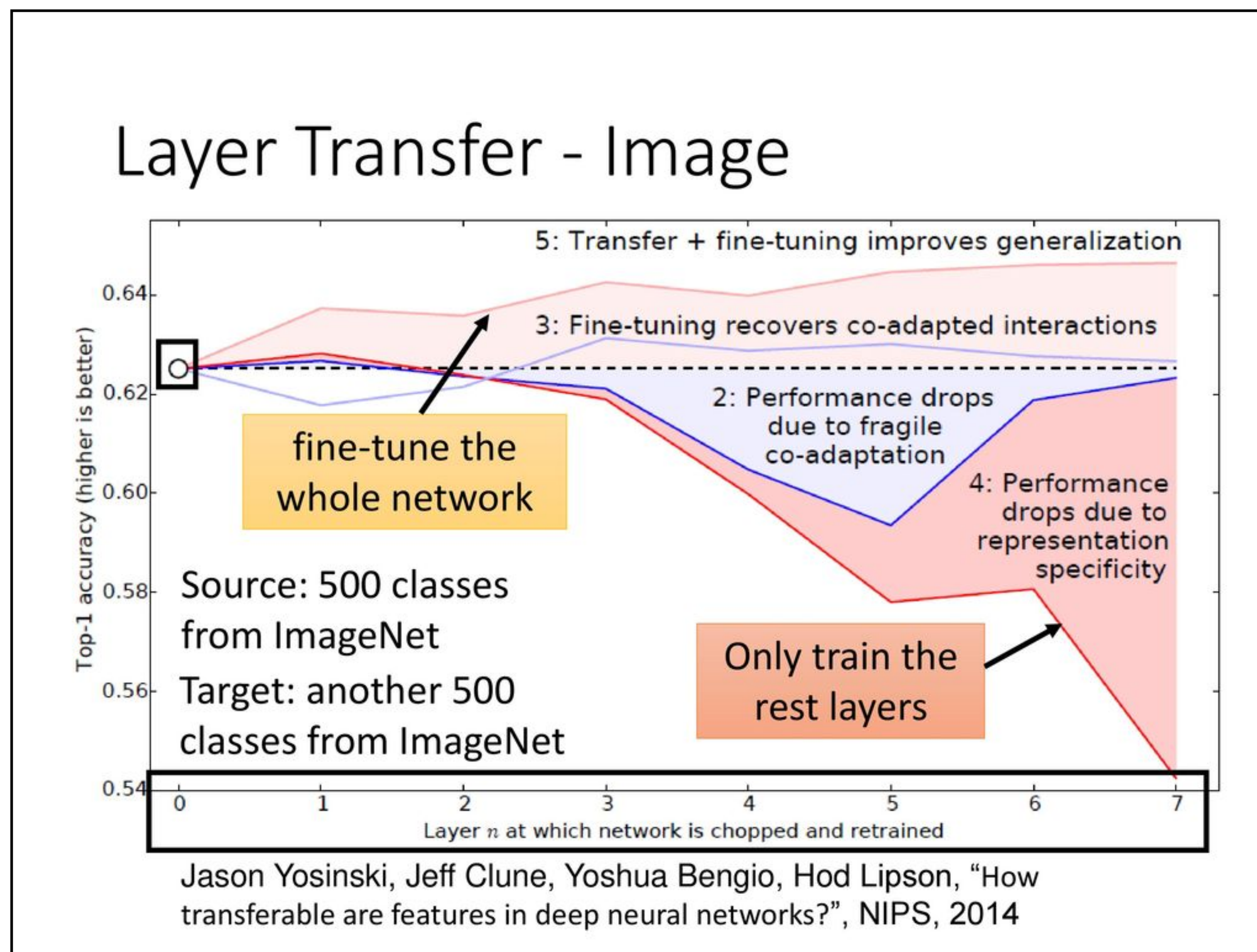
Transfer learning: pre-training + fine-tuning

- You want to train a predictor for a complex recognition task, but suffer from lack of training data.
- A predictor for a different task has been successfully trained on a large dataset.
- The domains of the two tasks are similar.



Improving the pipeline

- Use the first layers of the network that implements the predictor for the other task.
- Add your layers on top
- Learn the network on your data, if necessary apply early stopping to prevent overfitting.



Improving the pipeline

Automatic hyperparameter optimisation

(1) **Grid search**

- Select a small set of values for each hyperparameter. For numerical parameters: pick values on a logarithmic scale
- Train a model for each joint specification in the Cartesian product. Choose the combination with the smallest validation error.

The computational cost grows exponentially with the number of hyperparameters.

(2) **Random search**

- Define a marginal distribution for each hyperparameter.
- Sample random combinations from the product distributions.
- Train a model for each sampled combination.
- Choose the combination with smallest validation error.

Random search can be exponentially more efficient than grid search if several hyperparameters do not significantly affect the performance measure.