I2DL Summary - Zusammenfassung Introduction to Deep Learning

Introduction to Deep Learning (Technische Universität München)

Introduction to Deep Learning

SS 19

Machine Learning Basics

Unsupervised Learning

- no label or target class
- discover properties of the structure of data
- clustering (k-means, PCA)

k-nearest Neighbors

- training: store all data samples
- classification: most common class of the k-nearest (L2-distance) neighbors

Cross-Validation

- split data into training and validation set (usually 80/20)
- k-fold Cross-Validation: split data into k folds, use each subset exactly once as validation set, with the rest being the training set

Linear Regression

- find a linear model that explains a target y given the inputs X
- $\hat{y} = wx + b$

Logistic Regression

- linear regression with logistic (sigmoid) activation function
- outputs value from 0…1: can be interpreted as a probability → can be used for binary classification
- quadratic function

Loss/Objective/Energy Function

Measures how good the estimation is and tells the optimizer how to improve it.

Optimization

Changes the model in order to improve (minimize) the loss function.

Linear Least Squares

- approach to fit a mathematical model to the data, minimizing MSE
- convex problem: there exists a unique closed-form solution
- MSE in matrix notation: $MSE = (X\theta - y)^T(X\theta - y)$

Neural Networks as Computational Graph

- nodes are computations
- edges connect nodes
- directional (always forward!)
- organized in layers

Benefits:

- all small gradient computations can be parallelized → faster
- hard to get the analytical derivative of such complex functions

Loss Functions

L1 Loss

- sum of absolute error
- robust (to outliers)
- costly to compute
- optimum is the mean

L2 Loss

- sum of squared error
- prune to outliers
- compute-efficient
- optimum is the mean

Mean Squared Error

$$MSE = \frac{1}{N}\sum (t-p)^2$$

Binary Cross-Entropy Loss (Log Loss)

$$L(\hat{y}_i, y_i) = -\,y_i\,\log \hat{y}_i - (1-y_i)\,\log(1-\hat{y}_i) \quad \rightarrow \quad \begin{aligned} L(\hat{y}_i,1) &= -\log \hat{y}_i \\ L(\hat{y}_i,0) &= -\log(1-\hat{y}_i) \end{aligned}$$

Categorical Cross-Entropy Loss for Softmax Classifiers (Negative Log-Likelihood)

$$L(\hat{y}, y) = -\log(\hat{y}) \quad \text{where} \quad \hat{y} = \frac{e^{s_y}}{\sum_i e^{s_i}}$$

s_y : the computed score of the **correct** class

$$\frac{\partial L}{\partial s} = \hat{y} - y$$

(same derivative for sigmoid)

Hinge Loss (Multiclass SVM Loss)

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Hinge loss saturates at some point (when it has learned a class "well enough"), while Softmax always wants to improve!

Regularization

Any strategy that aims to lower the validation error while increasing the training error.

Append a regularization term to the cost function:

$$J(\theta) = \frac{1}{N}\sum_i L_i + \lambda R(\theta)$$

L1: sum of the absolute weight values

- focuses attention on key features
- enforces sparsity

$$R(\theta) = \sum_i |\theta_i|$$

L2: sum of the squared weights

- takes all information into account when making decision
- tries to distributes the weights (make all weights roughly the same)
  → makes every neuron participate → higher chance of generalization

$$R(\theta) = \sum_i \theta_i^2$$

L1 and L2 are weight regularization techniques!

Weight Decay

Adds a term to the update step equation that slightly reduces the weights (magnitude) at each step.

- penalizes large weights
- improves generalization

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta L(\theta_k, X, Y) - \lambda\theta^2$$

Data Augmentation

Why is data augmentation helpful?

- increases the amount of training data
- a classifier has to be invariant to transformations (translation, rotation, etc.)

Examples:

- image cropping (random crops at test time, fixed set of crops at test time)
- mirroring
- color manipulations

Early Stopping

Automatically stop training when the generalization gap grows.

Bagging and Ensemble Methods

Bagging: ensemble technique that combines the predictions of multiple models to form a final prediction.

Dropout

- disable a random set of neurons
- intuition: half the network = half the capacity
- can be considered as model ensemble (2 models in one, each trained with a different part of the data)
- reduces capacity of the model → larger models → more training time
- **no dropout at validation and test time!**

Regularizing effects of dropout:

1. adds noise to the learning process and training with noise has a regularizing effect
2. model ensemble: multiple models in one that share weights → each model regularizing the others

Why simply adding more layers doesn't help

- no structure
- it's just brute force
- optimization becomes harder
- performance plateaus / drops

Optimization & Learning

Gradient Descent

Update Rule:

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta L(\theta_k, X, Y)$$

1. **Batch/"Vanilla" Gradient Descent:** calculate mean gradient of the whole data set
2. **Stochastic Gradient Descent:** approximates the true gradient by a gradient of a single example
3. **Mini-Batch Gradient Descent:** approximates the true gradient by the mean gradient of a mini-batch

Problems:

Gradient is scaled equally across all dimensions

→ cannot independently scale directions

→ small learning rate (slower learning) needed to avoid divergence

→ not useful for linear systems!

Gradient Descent with Momentum

Increases the step size when a sequence of gradients all point to the same direction.

→ less "horizontal" *jiggling and* faster down into the minimum.

Hyperparameter β: "friction" / accumulation rate (often: $0{,}9$)

$$v_{k+1} = \beta\, v_k + \nabla_\theta L(\theta_k)$$
$$\theta_{k+1} = \theta_k - \alpha v_{k+1}$$

Nesterov's Momentum

$$\tilde{\theta}_{k+1} = \theta_k - v_k$$
$$v_{k+1} = \beta\, v_k + \nabla_\theta L(\tilde{\theta}_{k+1})$$
$$\theta_{k+1} = \theta_k - \alpha v_{k+1}$$

1. big step in direction of the previous gradient
2. measure the gradient of where you end up
3. step in corrected direction

RMSProp (Root Mean Squared Prop)

Divides the learning rate by an exponentially-decaying average of squared gradients.

→ dampens oscillations in high-variance directions → less oscillations, less likely to diverge

→ learning rate can be increased → faster learning

Hyperparameters: α (tuned), β (often: $0{,}9$), ϵ (often 10^{-8}, prevents division by 0)

$$s_{k+1} = \beta\, s_k + (1-\beta)(\nabla_\theta L)^2$$
$$\theta_{k+1} = \theta_k - \frac{\alpha}{\sqrt{s_{k+1}} + \epsilon} \nabla_\theta L$$

"second momentum"

Adam (Adaptive Moment Estimation)

Combines momentum and RMSProp → exponentially decaying mean and variance of gradients

Hyperparameters:

- α (tuned): initial learning rate
- β_1 (often: $0{,}9$): exponential decay rate for the first moment
- β_2 (often: $0{,}999$): exponential decay rate for the second moment
- ϵ (often 10^{-8}): just to prevent divisions by 0

$$m_{k+1} = \beta_1 m_k + (1-\beta_1)\nabla_\theta L(\theta_k)$$

first momentum: mean of gradients

$$v_{k+1} = \beta_2 v_k + (1-\beta_2)(\nabla_\theta L(\theta_k))^2$$

second momentum: variance of gradients

$$\theta_{k+1} = \theta_k - \alpha \frac{\hat{m}_{k+1}}{\sqrt{\hat{v}_{k+1}} + \epsilon}$$

parameter update

m_0 and v_0 are initialized as a vectors of zeros → bias towards zero!
To counteract this bias, the moment updates are usually bias-corrected with:

$$\hat{m}_{k+1} = \frac{m_k}{1-\beta_1}$$

$$\hat{v}_{k+1} = \frac{v_k}{1-\beta_2}$$

Newton's Method

Approximate the function by a second-order Taylor series expansion (includes curvature).
Exploits also the curvature to take a more direct route down into the minimum.
Method: differentiate and equate to zero (no learning rate)
Network parameters: n, elements in Hessian: n^2, update step needs requires matrix inversion with $O(n^3)$
Only works well for full batch updates (in this case faster than gradient descent), never the case in DL
In convex problem: finds minimum in one step
→ not computationally feasible with large datasets

BFGS and L-BFGS (Quasi-Newton Methods)

"Broyden-Fletcher-Goldfarb-Shanno algorithm"
Approximates the inverse of the Hessian → $O(k^2)$
Limited memory L-BFGS: $O(k)$

Gauss-Newton

Approximates the Hessian.

Levenberg

Damped version of Gauss-Newton, interpolates between Gauss-Newton ($\lambda = 0$) and gradient descent ($\lambda = 1$).

Levenberg-Marquardt

Instead of plain gradient descent for large λ, scale each gradient component along the curvature
→ faster convergence in components with small gradient

Newton variations (Newton, Quasi-Newton, GN, LM, etc.) would converge much faster than first order methods but they don't work on mini-batches (due to the stochasticity)!

Which Optimizer for which Dataset?

- for small (e.g. 10.000 examples) datasets with low redundancy, use a full-batch method, e.g. L-BFGS.
- for big, redundant datasets, use mini-batch methods (Adam, etc.)

Training vs. Learning

Training: find optimal weights for a given training set (training set)
Learning: generalization to unknown data (validation/test set)

Purpose of Data Sets

- **training set:** for training the model
- **validation set:** hyperparameter optimization, checking generalization progress
- **test set:** only use **once** at the very end to check model performance

When the validation error is lower than the training error:

1. most likely: bug in the implementation
2. bad data distributions: e.g. val set only contains images of the class that the model predicts best

Why not tuning hyperparameters on the test set? The model may overfit the test set and not generalize well to unseen data. Using the test set to estimate performance results would produce an overestimate.

Recipe

- first, **overfit** to single sample, then to a few.
- try a **small architecture** first (verify that it's learning something)
- **high training error** → bigger model, longer training, different architecture
- **high validation error** → more data, regularization, different architecture

Hyperparameter Search

- e.g. manual, grid search, random search
- grid/random: computationally very expensive, but can be highly parallelized
- if a combination starts performing better than others it usually stays better
  → stop the others to save computation

Activation Functions

Sigmoid

- saturates on outer regions
- always positive outputs → gradients are either all positive or all negative → "zig-zag problem"
- good gradients around 0
- small gradients on outer regions
  → heavily dependent on initialization
  → vanishing gradients

$\sigma(x) = \frac{1}{1+e^{-x}}$	$\frac{\partial \sigma}{\partial x} = \sigma(1-\sigma)$

tanh

- zero-centered → solves "zig-zag problem"
- also saturates
- used in recurrent nets

ReLU

- large and consistent gradient
- doesn't saturate
- fast convergence
- simple to compute
- dead ReLU problem: positive bias makes it likely that neurons stay active

Leaky ReLU

$$f(x) = \begin{cases} 0{,}01x, & \text{if } x < 0 \\ x, & \text{otherwise} \end{cases}$$

- solves dead ReLU problem
- used in GANs

Parametric ReLU

$\max(\alpha x, x)$

- α is a learned parameter
- unstable

Maxout

- learns a piecewise linear function
- returns the maximum of its inputs
- doubles the number of parameters

Weight Initialization

Problem of all-zero weights: all units will compute the same gradient → symmetry breaking

Small Random Gaussian Weights

- safe place for tanh/sigmoid → nice gradients from activations
- with sigmoids, *too small* weights essentially remove non-linearities
- small weights multiply with small weights → vanishing gradients

Big Random Gaussian Weights

- everything is saturated → vanishing gradient from activations

Xavier Initialization

- assumption: input is normalized to zero mean
- Gaussian variance dependent on the number of neurons in the layer
- sample weights from a zero mean Gaussian distribution with variance 1/n
- still vanishing gradients with ReLU as it kills half of the inputs → double variance to 2/n
- still, gradients die out in very deep nets

Batch Normalization

Goal: unit Gaussian inputs → unit Gaussian outputs, i.e. keep the activations alive (no vanishing gradients)

$$\hat{x}_k = \frac{x_k - \mu(x_k)}{\sigma(x_k)}$$

1. normalize input

$$y_k = \gamma_k \hat{x}_k + \beta_k$$

2. scale and shift

- BN layers are placed between FC and activation
- all biases before a BN layer will get cancelled out by BN
- more stable gradients
- makes deep nets easier to train
- makes larger range of hyperparameters work
- allows for higher learning rates
- reducing dependence on good initialization

The network can learn to undo it by scaling and shifting the normalization:

- beta must converge to the mean of the input
- gamma must converge to square root of the variance of the input

The normalization itself can not be learned because it has no gradients to work with.

At test-time: mean and variance are an exponentially weighted running average of training mini-batches.

Drawbacks: higher batch-sizes needed (works worse on small mini-batches)

Convolutional Neural Networks

Types of Layers: convolutional, pooling (max / average), fully connected, also upsampling, transpose convolutions (e.g. for segmentation)

Convolutional Layer

Extracts features from the input.

Hyperparameters: F: filter size, S: stride, P: padding

Output dimension (excl. channels):	$Output = \left(\frac{N - F + 2P}{S} + 1\right)^2$, n: width/height of input
Number of weights:	$F \cdot F \cdot C_{in} \cdot C_{out} + C_{out}$ (biases)
Number of biases:	1 per filter
Weights per filter:	$F \cdot F \cdot C_{in}$
Valid Convolution:	no padding, drops last convolution if dimensions don't match
Same Convolution:	$P = \frac{F-1}{2}$ → keeps the size of activation maps the same
Full Convolution:	set padding such that the whole filter sees the input end to end

Padding

- avoids activation map size getting too small
- corner pixels are only used once → more often with padding
- zero padding: padding with zeros (≠ **no padding**)

Pooling Layer

Progressively reduce the size of the representation to reduce the amount of parameters.

Intuition: once a feature has been found, its exact location isn't as important as its rough location relative to other features.

Hyperparameters:

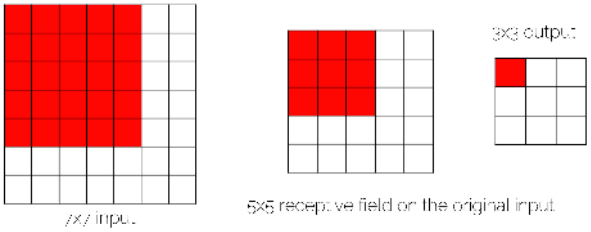
- K: kernel-size
- S: stride

Output dimension: $\left(\frac{N - K}{S} + 1\right)^2$

- typically applied after a conv layer
- max-pooling: downsamples the extracted features → picks the strongest activation in a region
- max-pooling: creates small local spatial/translational invariance (only within the pool region)
- max-pooling: some precision is lost, but it enables focus on the most important features
- progressively reduces the amount of parameters to save computation and reduce overfitting

Receptive Field

The spatial extend of the connectivity of a filter.
In other words: the information that went into computing that filter.



Architectures

LeNet

LeCun et al., 1998
Dataset: digit recognition, greyscale 32x32x1 → 10 classes
Activations: tanh/sigmoid
Architecture: [Conv + Avg. Pool] x2 → Conv → FC x2
Parameters: 60k

AlexNet

Dataset: ImageNet, RGB 227x227x3 → 1000 classes
Architecture:

- first filter with stride 4 to reduce size significantly → made training it manageable
- similar to *LeNet*, but uses max pooling and ReLU
- same convolutions
- two FC at the end

Parameters: 60M

VGG

Idea: much simpler architecture!

- conv: always 3x3 filters, stride 1, same convolutions
- max pool: 2x2 filters, stride 2

Architecture:

- conv x2/x3 → max pool (extract features slowly via 2 or 3 conv layers before pooling)
- number of filters doubles with each conv block

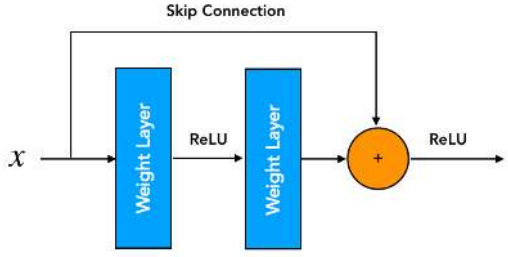
Models: VGG-16/19 (16/19 layers **with weights**)
Parameters: 138M

ResNet

A network that stacks residual blocks → enables far deeper nets without vanishing/exploding gradients because larger gradients get propagated back.

Residual block: sum up the input of a layer via a skip connection to the output of a layer 2-3 levels deeper, *before* the activation function.
→ convolutions in residual blocks need to preserve dimensions (same convolutions)

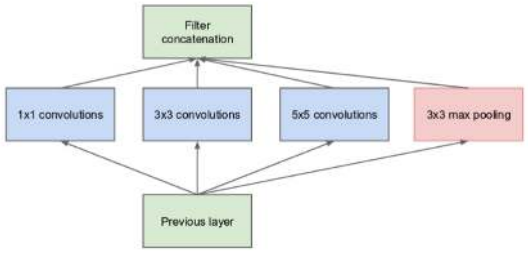
Residual blocks can not hurt performance! The "worst" case is that a residual block learns the identity and only the skip connection is used → thus simply ignoring the layers in the block



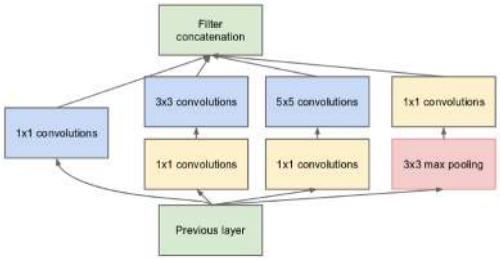
Plain Networks: performance starts to degrade once the network gets too deep
ResNets: performance keeps increasing, the deeper the network gets
Skip connections can be seen as making the number of layers dynamic. The network can simply learn to skip layers (learn identity) that do not benefit performance.

Inception Layer

Idea: use all possible filter sizes in the same layer, instead of trying to find the best filter size.
Layers in a naïve inception block: 1x1 conv, 3x3 conv, 5x5 conv, 3x3 max pooling (image a)
Concatenation: stack all activation maps (convolutions must preserve dimension)
→ too many channels
→ dimensionality reduction: insert 1x1 convolutions to reduce the depth (image b)



(a) Inception module, naïve version



(b) Inception module with dimension reductions

Fully Convolutional Networks

- only contains conv layers, no FC layers.
- e.g. a FC layer can be replaced by a 1x1 convolution → invariance to input image size!
- enables returning to image space by upsampling, e.g. for semantic segmentation with pixel-wise predictions

Types of upsampling:

1. interpolation: few artifacts
2. transposed convolution: unpooling + learned conv filter

1x1 Convolutions

Can be used to change the number of filters without changing image dimensions
→ dimensionality reduction to reduce complexity
e.g. used in inception layers to make it computationally manageable

Transfer Learning

Idea: use a pre-trained network and fine-tune it on a dataset.

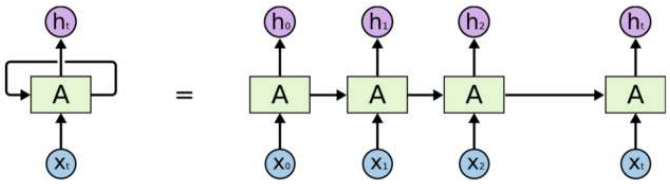
Why does it work?
For similar tasks (e.g. image classification), the features that a network learns likely also work on a different data set, especially since low level features like edges and corners apply to most images.

Advantage: don't need to learn early low-level conv weights that don't care about exact image contents
Requirements: input format is exactly the same

Fine-tuning:

- freeze most layers of the pre-trained net and only optimize e.g. the output layer
- the larger the dataset, the more of the pre-trained layers can be updated
- pre-trained layers are only updated with a very small learning rate → different learning rates

Recurrent Neural Networks



https://bit.ly/1iaBaLH

Hidden state is always the same → same parameters for each time step!

Unrolling / Unfolding

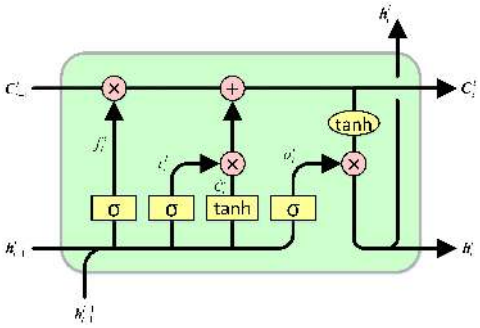
Converting the RNN (acyclic graph) into a directed graph (see image above)

Long-term Dependencies

Problem: long chains → vanishing gradients with small weights
Vanishing gradients: recent inputs have way more influence than earlier inputs → "forgetting"

Long Short Term Memory Networks (LSTM)

Idea: cell state that can maintain information in memory over a long time, "skip-connections for RNNs"
LSTM-Unit: cell state that is modified by a number of gates via element-wise operations



https://bit.ly/1iaBaLH

Forget gate	decides when to erase the cell state (sigmoid = 1 → keep, 0 → forget)	$f_t = \sigma(\theta_{xf} x_t + \theta_{hf} h_{t-1} + b_f)$
Input gate	decides which values will be updated	$i_t = \sigma(\theta_{xi} x_t + \theta_{hi} h_{t-1} + b_i)$
Output Gate	decides which values will be outputted	$o_t = \sigma(\theta_{xo} x_t + \theta_{ho} h_{t-1} + b_o)$
Cell update		$g_t = \tanh(\theta_{xg} x_t + \theta_{hg} h_{t-1} + b_g)$
Cell	transports information through the unit	$C_t = f_t \odot C_{t-1} + i_t \odot g_t$
Output		$h_t = o_t \odot \tanh(C_t)$