PROGRAMMING IN PYTHON II

Data Loading and Types of Data

Michael Widrich
Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Institute for
Machine Learning

Outline

MOTIVATION

Recap

- Last Unit, we learned that we
    1. want our model to generalize to unseen data
    2. need i.i.d. data to get an estimate for the generalization of our model (testset)
    3. can use clustering methods to inspect our data and search for potential issues
    4. might have to preprocess and normalize our data before feeding it to our method

Goal

- We want to feed our dataset to our model
- For this, we will learn
  1. which types of (statistical) data exist
  2. what our data need to look like for gradient-based methods (e.g. neural networks (NNs))
  3. about bottlenecks for loading data
  4. PyTorch Dataset and DataLoader

TYPES OF DATA

Typical ML point-of-view

- In ML we can represent our samples by vectors of feature values (=feature vectors) of length d

$$\mathbf{x} = (x_1, \ldots, x_d)T$$

  - E.g.: Representing dogs by their height and weight would require 2 feature values (i.e. $d = 2$)

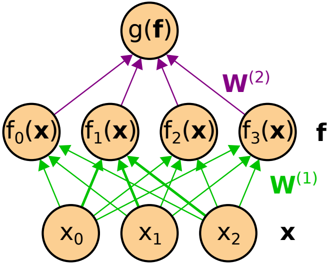
- We assume our feature vectors to be from a set/space X

$$\mathbf{x} = (x_1, \ldots, x_d)^T \in X$$

- If X is finite set of labels, we speak of *categorical variables/features*
- If $X = \mathbb{R}$, real interval, etc., we speak of *numerical variables/features*

Fully-connected feed-forward NN

- Standard fully-connected feed-forward NN (FFNN)



$$\mathbf{W}^{(1)} = W^{(1)}_{0\ldots3,0\ldots2}$$

$$f_1(x) = a(W^{(1)}_{1,0} * x_0 + W^{(1)}_{1,1} * x_1 + W^{(1)}_{1,2} * x_2) = a(\textstyle\sum_j^n (W^{(1)}_{1,j} \cdot x_j))$$

  - Weights $\mathbf{W}$ are adjusted such that $g(\mathbf{x}; \mathbf{W}) \xrightarrow{training} target$
  - a is an activation function, e.g. sigmoid, relu, selu, ...

Types of data

- We have 3 different types of data (in the statistical sense)
  1. Numerical data
  2. Categorical data
  3. Ordinal data
- In practice, we (usually) use the `float` datatype for our NN computations

$\rightarrow$ We need to represent our data as `float`

NUMERICAL DATA

Numerical data – Theory

- Data with quantitative meaning
- Continuous data
  - Measurements that cannot be counted (uncountably infinite)
  - Described using intervals on real number line
  - Example: Any real number in range $[0, 10]$
  - E.g. size of a leaf on a plant
- Discrete data
  - Countable data (countably finite or infinite)
  - Example (finite): $0, 1, 2, \ldots, 10$
  - Example (infinite): $0, 1, 2, \ldots, \infty$
  - E.g. number of leafs on a plant

Numerical data – Practice (1)

- We want to represent a numerical data value as a `float` value
- Problem: `float` values have limited number of bits

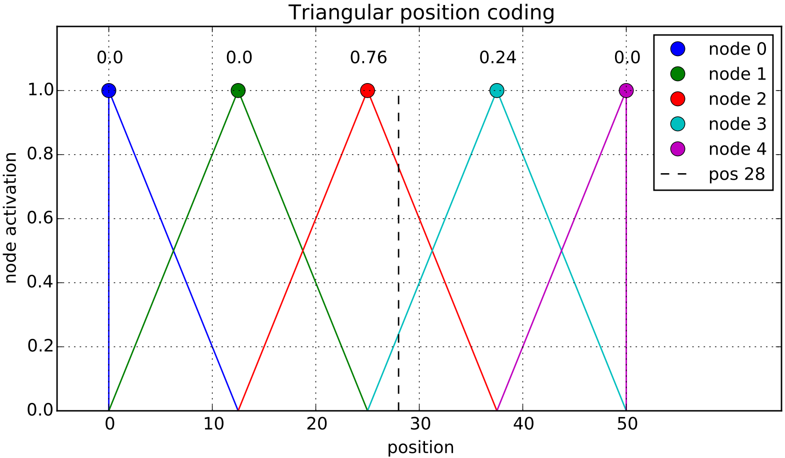
Numerical data – Practice (1)

- We want to represent a numerical data value as a `float` value
- Problem: `float` values have limited number of bits
- Approximate (quantize) numerical data:
  - Cap value ranges (to finite)
  - Lose precision (limited number of bits)
  - Focus on value ranges that are important for task
    - Common: Clip, square, logarithm, square root, sigmoid, tanh
    - Requires prior knowledge

Numerical data – Practice (2)

- Problem: Especially for discrete data with large value ranges:
  - Loss of precision makes task unsolvable
  - Learning to focus on precise values is difficult for NN
    - Has to adjust bias weights very precisely
- Solution: Encode the numerical value in multiple input units
  - Each unit spans a range of values (e.g. triangles, gaussians)
  - Common: Encoding of position or time

Discrete data – Triangle encoding



CATEGORICAL DATA

Categorical data – Theory

- Qualitative data
- No mathematical meaning
- Mathematical operations (e.g. $\sum$) do not make sense
  - Example: “Dog”, “Rat”, “Cat”

Categorical data – Practice (1)

- Assume we consider n different categories
- We could represent each category as integer value
  - Requires n different integer values, one for each category
  - We could encode these integer values as `float`
  - Example: “Dog”$= 0$, “Rat”$= 1$, “Cat”$= 2$

Categorical data – Practice (1)

- Assume we consider n different categories
- We could represent each category as integer value
  - Requires n different integer values, one for each category
  - We could encode these integer values as `float`
  - Example: “Dog”$= 0$, “Rat”$= 1$, “Cat”$= 2$
- Problem: Our method (e.g. NN) performs mathematical operations on the input values!
  - We would introduce new (probably false) information
  - The NN would first have to learn to ignore this false information
  - Example: In our ranking, “Dog” $<$ “Rat’ and “Rat” $\cdot 2 =$ “Cat”

  $\rightarrow$ Not suitable for us

Categorical data – Practice (2)

- Solution: Represent a categorical feature as binary vector $\mathbf{v}_{0...n}$
  - Categorical data with n different values is enumerated from $0 \ldots n$
  - $v_i = 1$ if category i is true, otherwise $v_i = 0$
  - Each element in the vector represents one category $\rightarrow$no false information!*
- Example:
  - Possible values: “Dog”, “Rat”, “Cat” ($n = 3$)
  - Sample is “Dog” $\rightarrow \mathbf{v} = (1, 0, 0)^T$
  - Sample is “Cat” $\rightarrow \mathbf{v} = (0, 0, 1)^T$

Categorical data – Practice (2)

- Solution: Represent a categorical feature as binary vector $\mathbf{v}_{0 \ldots n}$
  - Categorical data with n different values is enumerated from $0 \ldots n$
  - $v_i = 1$ if category i is true, otherwise $v_i = 0$
  - Each element in the vector represents one category $\rightarrow$no false information!*
- Example:
  - Possible values: “Dog”, “Rat”, “Cat” ($n = 3$)
  - Sample is “Dog” $\rightarrow \mathbf{v} = (1, 0, 0)^T$
  - Sample is “Cat” $\rightarrow \mathbf{v} = (0, 0, 1)^T$
- *) Only applies if information about order in feature vector is not used

Categorical data – Practice (3)

- Mutually-exclusive categories:
  - One-hot feature vector
  - Only one element in feature vector is 1, others are 0
  - Feature vectors are typically sparse
- Categories include combinations (e.g. “Dog” and “Cat”):
  - Can be encoded/embedded in binary feature vector with multiple 1-entries per sample
  - NN does not have to learn that e.g. feature 5 is a combination of feature 2 and feature 26
- Additional information (e.g. measurement certainty)
  - Values in binary feature vector can be scaled to increase/decrease signal strength of input

ORDINAL DATA

Ordinal data – Theory

- Mix of numerical and categorical data
- Ranking between categories exist but distance is unknown
- Example: “small”, “medium”, “large”
  - Ranking “small” $<$ “medium” $<$ “large” exists
  - Distance is unclear (“small”$+$“small”$=$“medium”?)

Ordinal data – Practice (1)

- Apply our approach from categorical data
- Sort features in $\mathbf{v}$ according to ranking
- Fully-connected feed-forward NN
  - No initial awareness about order of features in $\mathbf{x}$
  - FFNN has to learn ranking by itself
  - Possible but not efficient
- Better: Use methods that naturally take ranking/hierarchy in feature vector into account
  - FFNN vs. CNN, RNN, attention, graph-NNs, ...
  - Allows for additional information via 1D, 2D, 3D, nD feature matrices or graph representations

Ordinal data – Practice (2)

- Especially in natural language processing (NLP):
  Use learned or fixed embedding of features
- Features are projected to feature space with better properties for NN training
- Simple approach: Random combinations of existing features as new features
- Better: Include prior knowledge of relationships within categories
  - Good combinations of categories easier accessible for NN
  - Example: Handcrafted embedding, pre-trained embeddings (NLP), dynamically learned embeddings

COMBINING TYPES OF DATA

Combining types of data (1)

- Naive approach: Concatenate different sets of features to one $\mathbf{x}$
  - Often done, easy, computationally cheap
  - (Sets of) features should be normalized individually across sample or dataset
- Problems:
  - Larger sets of feature dominate input
  - Different feature spaces have different suitable methods

Combining types of data (2)

- Concatenate outputs of dedicated sub-NNs or methods
- Each sub-NN or method processes separate feature set
- Pros:
  - Feature sets can be processed with suitable methods of different complexity
  - Using self-normalizing NNs (SNNs), processed feature sets will be normalized automatically
  - Number of output features of sub-NNs determine which feature set should be more prominent
    - Equal number of output f.s $\rightarrow$feature sets contribute similarly
- Cons: Increases complexity (chance to overfit) and computation demands

FEATURE DESIGN FOR NN

Feature design for NNs (1)

- NNs are universal function approximators
  - I.e. with enough units you can build any function

Feature design for NNs (1)

- NNs are universal function approximators
  - I.e. with enough units you can build any function
- Problem: You need to train it first to get there

Feature design for NNs (1)

- NNs are universal function approximators
  - I.e. with enough units you can build any function
- Problem: You need to train it first to get there
  - . . . and there is not guarantee that this will work

Feature design for NNs (1)

- NNs are universal function approximators
  - I.e. with enough units you can build any function
- Problem: You need to train it first to get there
  - . . . and there is not guarantee that this will work
- NN are typically trained with gradient-based methods
  - Weights will move across error surface to suitable values
- We need a (smooth) path from our initial weights to our target weights
  - Otherwise we will get stuck in local minima and/or need more samples

  $\rightarrow$ Bad feature design can make your training fail

Feature design for NNs (2)

- How complex does the function have to be to create the target outputs from the inputs?
  - Do weights need to be very precise to separate good from bad output?
  - Example: hash value space vs. pixel space

Feature design for NNs (2)

- How complex does the function have to be to create the target outputs from the inputs?
  - Do weights need to be very precise to separate good from bad output?
  - Example: hash value space vs. pixel space
- Can the network change from initial output to target output smoothly?
  - Need to flip signs of weights or make large jumps to overcome worse outputs?
  - Need to set weights to 0 to quickly down-weight large inputs? (0 activation $\rightarrow$no gradient/path)

Feature design for NNs (2)

- How complex does the function have to be to create the target outputs from the inputs?
  - Do weights need to be very precise to separate good from bad output?
  - Example: hash value space vs. pixel space
- Can the network change from initial output to target output smoothly?
  - Need to flip signs of weights or make large jumps to overcome worse outputs?
  - Need to set weights to 0 to quickly down-weight large inputs? (0 activation $\rightarrow$no gradient/path)
- Which information does the NN have to unecessarily encode from the inputs?
  - E.g. mean/std of pixel values, position, time, Δ of values (e.g. positions)

EXAMPLES: FEATURE DESIGN FOR NNS

Example: XOR (1)

- XOR (Exclusive Or) of two inputs x_0 and x_1:

$$xor(x_0, x_1) = \begin{cases} 1, & \text{if } (x_0 \text{ or } x_1) \text{ and not } (x_0 \text{ and } x_1) \\ 0, & \text{otherwise} \end{cases}$$

Example: XOR (1)

- XOR (Exclusive Or) of two inputs x_0 and x_1:

$$xor(x_0, x_1) = \begin{cases} 1, & \text{if } (x_0 \text{ or } x_1) \text{ and not } (x_0 \text{ and } x_1) \\ 0, & \text{otherwise} \end{cases}$$

- Task: Learn XOR with NN with activation function a_{relu}:

$$a_{relu}(v) = \begin{cases} v, & \text{if } v \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Example: XOR (1)

- XOR (Exclusive Or) of two inputs x_0 and x_1:

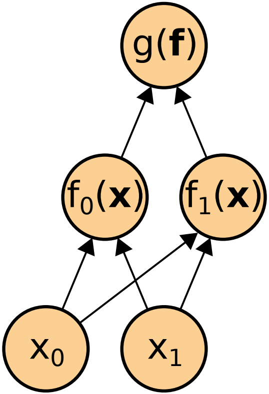
$$xor(x_0, x_1) = \begin{cases} 1, & \text{if } (x_0 \vee x_1) \wedge (\neg (x_0 \wedge x_1)) \\ 0, & \text{otherwise} \end{cases}$$

- Task: Learn XOR with NN with activation function a_{relu}:

$$a_{relu}(v) = \begin{cases} v, & \text{if } v \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

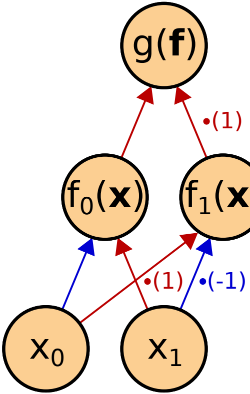
Example: XOR (2)

- Theoretically, this NN is enough for a solution:



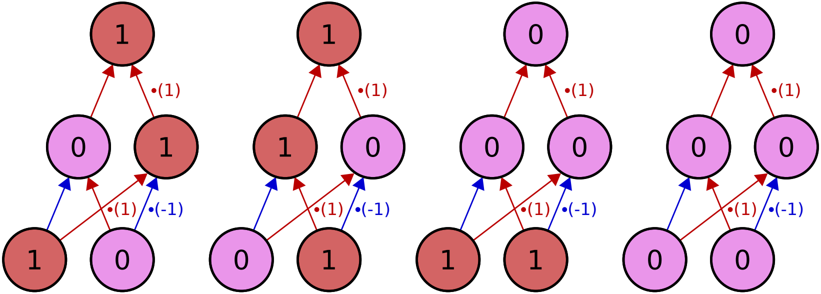
Example: XOR (3)

- Theoretically, this NN is enough for a solution:



Example: XOR (4)

- Theoretically, this NN is enough for a solution:



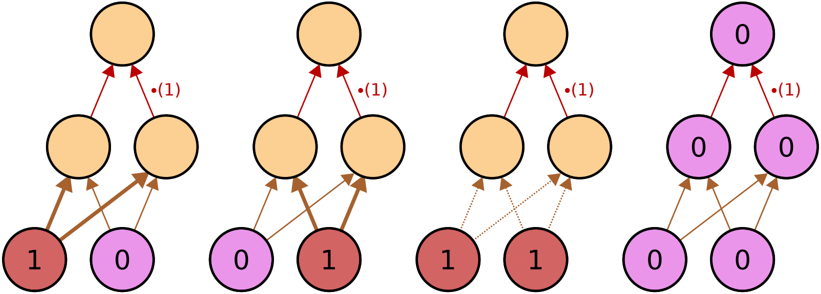
Example: XOR (5)

- But being able to represent the solution is not enough, we need to find it too!

→ Can we find the solution via gradient descent starting from random weights?

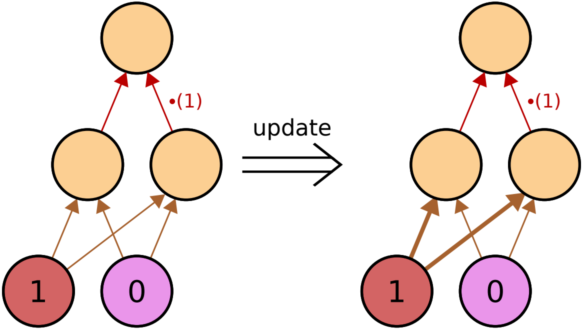
Example: XOR (6)

- This would be the updates for the weights for different inputs (bold: increase weight, dotted: decrease weight):



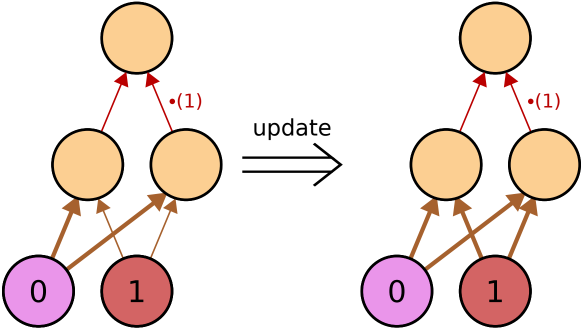
Example: XOR (7)

- Let's perform some consecutive update steps, given some samples:



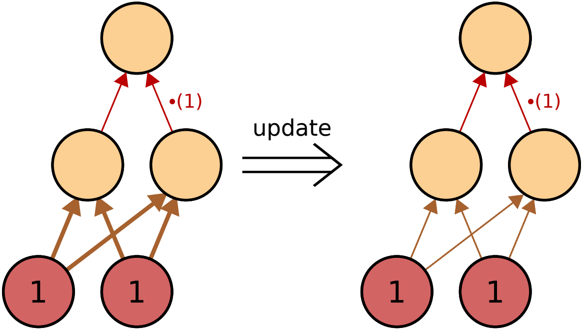
Example: XOR (8)

- Let's perform some consecutive update steps, given some samples:



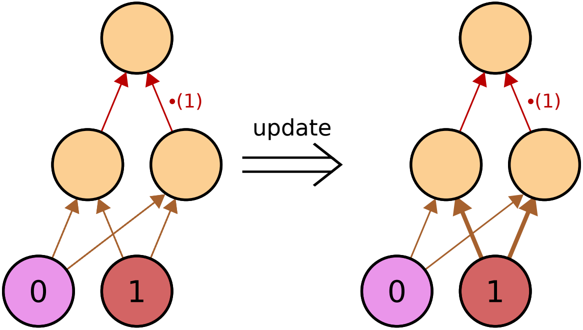
Example: XOR (9)

- Let's perform some consecutive update steps, given some samples:



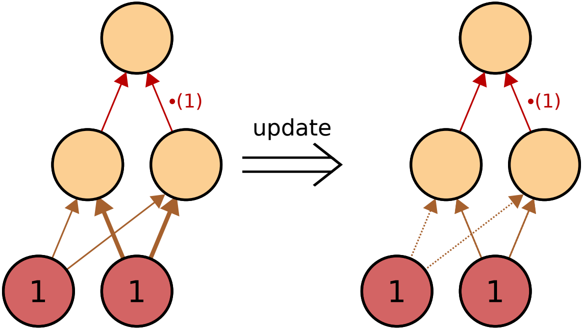
Example: XOR (10)

- Let's perform some consecutive update steps, given some samples:



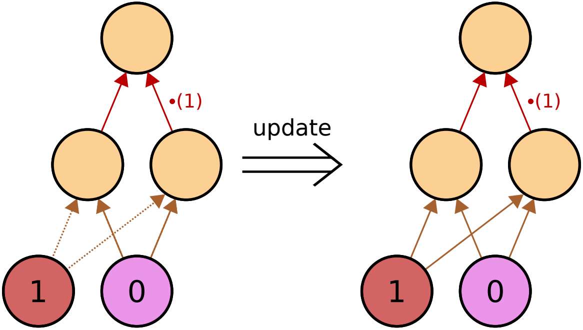
Example: XOR (11)

- Let's perform some consecutive update steps, given some samples:



Example: XOR (12)

- Let's perform some consecutive update steps, given some samples:



Example: XOR (13)

- We are not reaching our solution :(

Example: XOR (13)

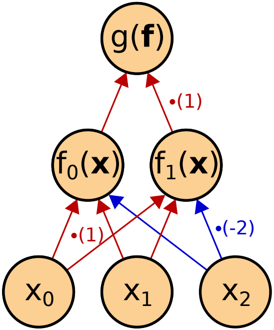
- We are not reaching our solution :(
- We could add more units to our NN and hope to have a close-to-solution weight combination in the initialized weights or apply other fancy techniques…

Example: XOR (13)

- We are not reaching our solution :(
- We could add more units to our NN and hope to have a close-to-solution weight combination in the initialized weights or apply other fancy techniques...
- ...or we could just use better feature design!
  - Add more input features as combinations of x_0 and x_1
  - E.g.: Add $x_2 = (x_0 \wedge x_1)$

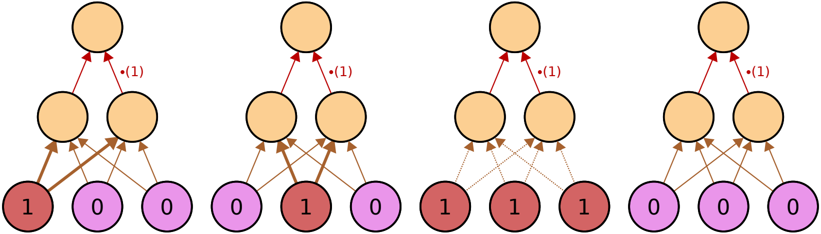
Example: XOR (14)

- With $x_2 = (x_0 \wedge x_1)$, a solution would be:



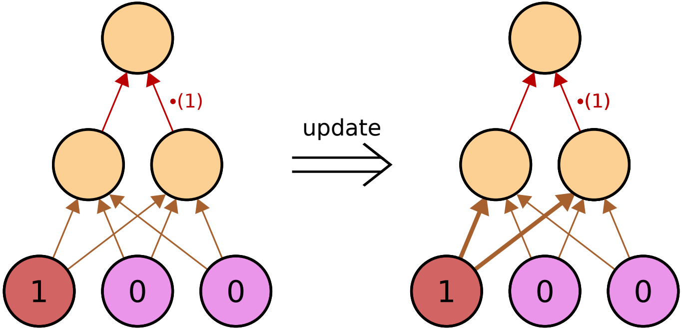
Example: XOR (15)

- Possible update steps for our network:



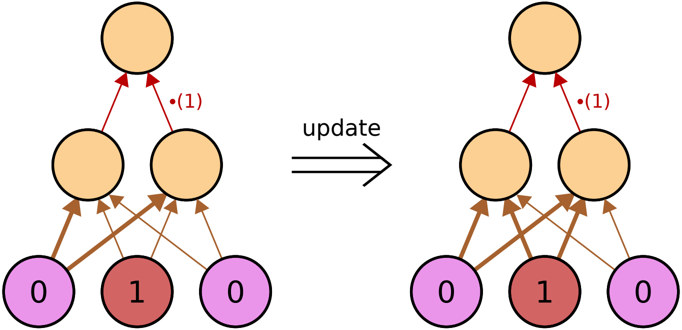
Example: XOR (16)

- Let's perform some consecutive update steps, given some samples:



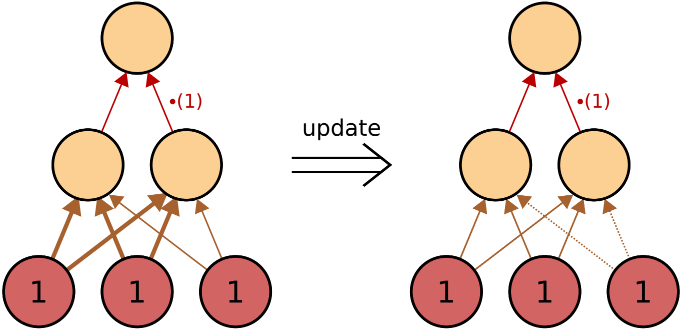
Example: XOR (17)

- Let's perform some consecutive update steps, given some samples:



Example: XOR (18)

- Let's perform some consecutive update steps, given some samples:

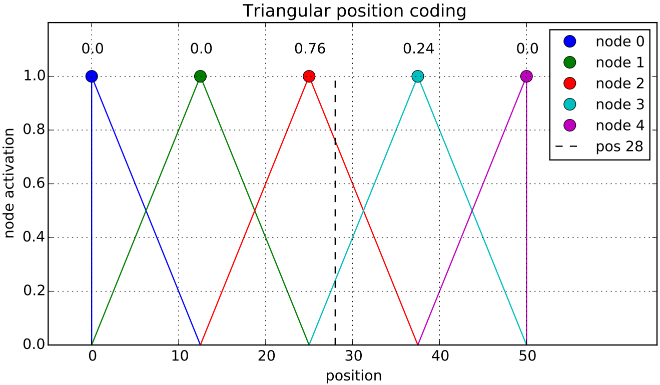


Example: XOR (19)

- We have found a solution easily just by adding 1 more input feature!

Example: Position encoding

- Note how the encoded ranges overlap by $1/2$
  - No cut-offs between positions
  - All positions have accumulated activation $1 \rightarrow$less initial bias



PYTHON II PROJECT

Types of data – Python II Project

- We are dealing with measurements of light (continuous),
- which have been quantized to `uint8` pixel values (discrete)
- We do not need high precision (do not need to single out precise brightness values)

→ Encoding a discrete pixel value as a `float` value is sufficient

- Could we include features that make prediction of the cropped-out image parts easier? (Assignment 2)

FURTHER READING

Further reading

- Courses and lecture materials in AI-study
- ML-, statistics-, image-/signal-processing courses at JKU
- *Pattern Recognition and Machine Learning* (C. Bishop)
- *Statistics For Dummies*: `https://www.dummies.com/education/math/statistics/types-of-statistical-data-numerical-categorical-and-ordinal/`
- *Dive into Deep Learning* (A. Zhang, Z. Lipton, M. Li, A. Smola): `https://d2l.ai/`

LOADING DATA: MINI-BATCH LEARNING

Mini-batch learning

- 3 types of utilizing samples to train NN:
  - Full-batch learning
    - All training samples used for 1 NN update
    - Gradients are averaged over samples
    - Smooth but weak gradients $\rightarrow$slow learning, overfitting
  - Online learning
    - 1 sample per weight update (shuffled samples)
    - Strong but not smooth gradients $\rightarrow$gradients might be contradicting
  - Mini-batch learning
    - b samples per weight update (shuffled samples)
    - Smooth gradients but strong enough to train fast with less overfitting
    - b is a hyperparameter that we need to optimize

LOADING DATA: BOTTLENECKS

Bottlenecks: Bandwith (1)

- General
  - Transfer as little data as possible
  - Prefer smaller data types
  - Make use of sparseness of data (compression and optimized computations)
- Network $\implies$ disk
  - Copy dataset to storage with fast connection to training device
- Disk $\implies$ RAM
  - Store dataset in RAM if possible
- RAM $\implies$ GPU memory
  - Only copy what you really need (input and output)
  - Prefer large coherent array vs. many small arrays

Bottlenecks: Bandwith (2)

- Example: One-hot feature vectors
- Setting:
  - We want to transfer many one-hot feature vectors to our GPU
  - Or mini-batch consists of 20 one-hot feature vectors of length 50

Bottlenecks: Bandwith (2)

- Example: One-hot feature vectors
- Setting:
  - We want to transfer many one-hot feature vectors to our GPU
  - Or mini-batch consists of 20 one-hot feature vectors of length 50
- Possible solutions:
  - Stack our feature vectors to one array before transfer
  - Only transfer indices of 1-elements and create full feature vector on GPU

    $\rightarrow$ Reduced from $20 \cdot 50 = 1,000$ to $20 \cdot 1 = 20$ bits!

Bottlenecks: Bandwith (2)

- Example: One-hot feature vectors
- Setting:
  - We want to transfer many one-hot feature vectors to our GPU
  - Or mini-batch consists of 20 one-hot feature vectors of length 50
- Possible solutions:
  - Stack our feature vectors to one array before transfer
  - Only transfer indices of 1-elements and create full feature vector on GPU
    - $\rightarrow$ Reduced from $20 \cdot 50 = 1,000$ to $20 \cdot 1 = 20$ bits!
  - (We only need 50 indices, we can use `uint8` to store indices)

Bottlenecks: Bandwith (3)

- Important:
  - Introduces possibility for bugs, always check if final sample on GPU equals sample on CPU
  - Check where the actual bottleneck is in you code (timeit module*)
  - Check if your approach is really faster
  - Performance-optimization is a trade-off $\rightarrow$how far do you need to go?

*) `https://docs.python.org/3/library/timeit.html`

Bottlenecks: Computation (1)

- Loading data often involves on-the-fly preprocessing and data augmentation
- Large datasets are typically stored with high compression and need to be decompressed
- For each NN update we need to load multiple samples (mini-batch learning)

$\rightarrow$ Considerable computational effort for loading data

Bottlenecks: Computation (2)

- Solution:
  Data loading is performed by multiple processes the background
- Background processes can prepare new minibatch during weight update
- Typicially done on CPUs
  - Access to large RAM with dataset
  - “Cheap” mass of CPUs
  - Exceptions: Embedding/preprocessing/data augmentation that require GPUs
- Multiprocessing often removes deterministic sample order
  - Less/no reproducibility

LOADING DATA: PYTORCH

PyTorch for loading dada

- PyTorch offers various tools for data loading*
- General: `torch.utils.data`**
  - Dataset classes, templates, unified interfaces
  - Loading data with support for background workers
- Relevant for vision-based tasks: `torchvision`***
  - Pre-processing and data augmentation pipe-lines
  - Pre-trained models
  - Standard public datasets

*) `https://pytorch.org/tutorials/beginner/data_loading_tutorial.html`
**) `https://pytorch.org/docs/stable/data.html`
***) `https://pytorch.org/docs/stable/torchvision/index.html`

PyTorch Dataset

- `torch.utils.data.Dataset`
- Dataset represented as class with standardized interface
- Derive your dataset class from `Dataset`
- Add your own method for reading a sample via `__getitem__()`
  - Should return 1 sample
  - Can include pre-processing and data augmentation
  - E.g. returns tuple of image as numpy array, label as `int`, and ID as `int`
- Provide number of samples in `__len__()`
- Can be wrapped by other classes, e.g. `torch.utils.data.Subset`

PyTorch DataLoader

- `torch.utils.data.DataLoader`
- Extracts minibatch of samples from `Dataset` instance
  - Supports shuffling and multiprocessing (not deterministic!)
  - Stacks samples to minibatch automatically (=batching)
  - Custom batching via `collate_fn` argument
  - Looping over `DataLoader` instance will return all samples of `Dataset` instance, one minibatch at a time

Classic usage

- Derive a class from `Dataset`: `MyDataset(Dataset)`
- Add `__getitem__()` (to read and return sample)
- Add `__len__()` (to return number of samples in dataset)
- Create dataset instance: `mydataset = MyDataset()`
- Create dataset splits via `torch.utils.data.Subset`:
  - `trainingset = torch.utils.data.Subset( mydataset, training_indices)`
- Create data loader (mini-batch size 16, using 4 background workers)

```
training_loader = DataLoader(trainingset,
batch_size=16, shuffle=True, num_workers=4)
```

- Loop over data loader to get minibatches
  - `for minibatch in training_loader: ...`

Hints

- Shuffling and multiprocessing is not deterministic (reproducability)
- Store indices of dataset splits in separate file and use `torch.utils.data.Subset` to create training, validation, and test set (reproducability)
- Disable shuffling in validationset/testset
- Warning: Avoid using DataLoaders in threads of `multiprocessing` (buggy), instead call Python scripts using `subprocess.call`/`subprocess.Popen`
- No need to use tensors in `Dataset`, you can stay in numpy
- Include sample ID in return from `__getitem__()` (debugging)