Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

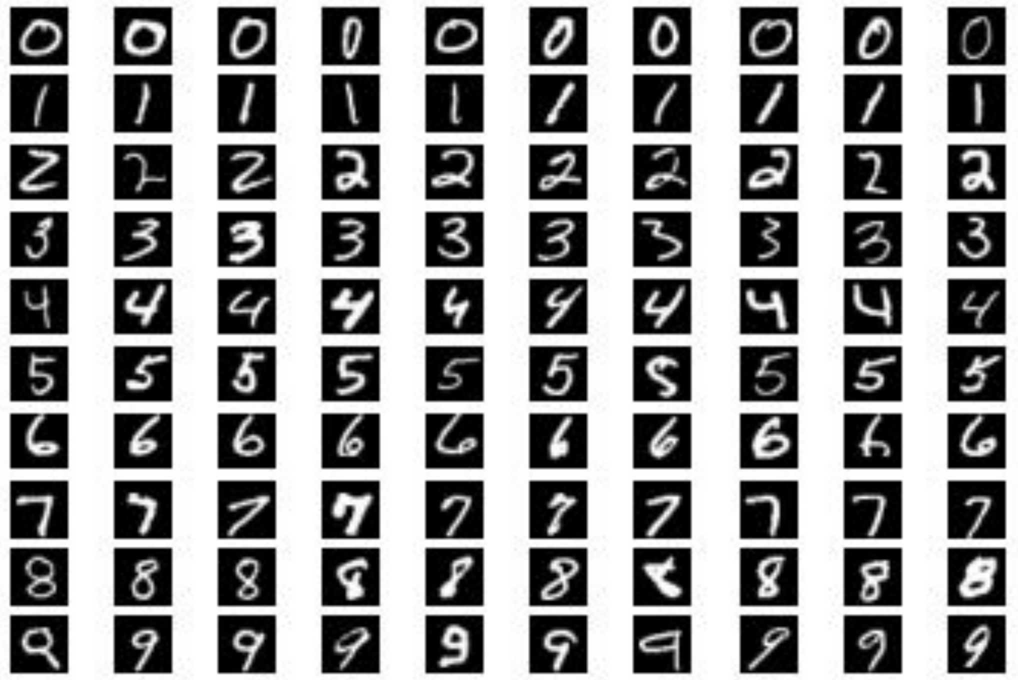
Deep Learning - Worksheet 2

Lecturer: Prof. Dr. Frank Noe
Tutorial: Tuesday 12 - 14, Andreas Krämer
Students: Hana Zupan, Ana Salgado, Coco Bögel, Esteban Lasso, Jim Neuendorf, Mert Efe and Adel Golghalyani

Agenda

- The MNIST dataset
- Dataset preparation & normalization
- Neural Network structure
- Easy example of a pure numpy network
- Loss-function: Hot one encoding
- How to train the model
- Validation of the model
- Hyperparameter Search / Hyperparameter Tuning

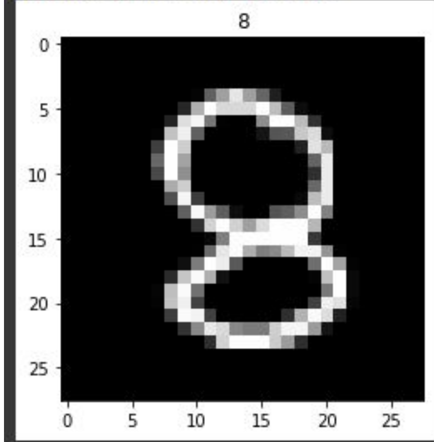
MNIST dataset



MNIST dataset - properties

```python
with np.load('/content/drive/MyDrive/Deep Learning/prediction-challenge-01-data.npz') as fh:
    data_x = fh['data_x']
    data_y = fh['data_y']
    test_x = fh['test_x']

# TRAINING DATA: INPUT (x) AND OUTPUT (y)
# 1. INDEX: IMAGE SERIAL NUMBER
# 2. INDEX: COLOR CHANNEL
# 3/4. INDEX: PIXEL VALUE
print(data_x.shape, data_x.dtype)
print(data_y.shape, data_y.dtype)

# TEST DATA: INPUT (x) ONLY
print(test_x.shape, test_x.dtype)

plt.imshow(data_x[0, 0], cmap='gray')
plt.title(data_y[0])
plt.show()
```

```
(20000, 1, 28, 28) float32
(20000,) int64
(2000, 1, 28, 28) float32
```



Dataset preparation & normalization

```python
# Transfor data to Tensor and create Train/Test-Set

data_x = torch.from_numpy(data_x)
data_y = torch.from_numpy(data_y)
X_train, X_test, y_train, y_test = train_test_split(data_x, data_y, test_size=0.2)
```

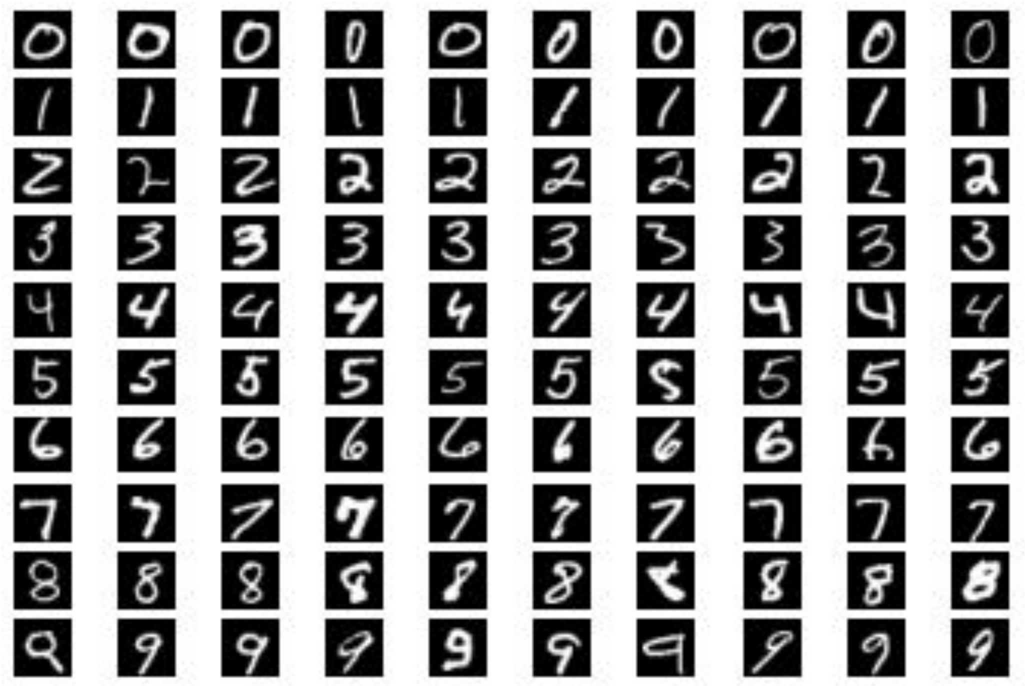
```python
# Reduce dataset to given split size, create some batches

BATCHSIZE = 100

X_train = torch.split(X_train, BATCHSIZE)
y_train = torch.split(y_train, BATCHSIZE)

X_test = torch.split(X_test, BATCHSIZE)
y_test = torch.split(y_test, BATCHSIZE)
```

```python
# normalization of the data
data_x = ( data_x - data_x.mean() ) / data_x.std()
```

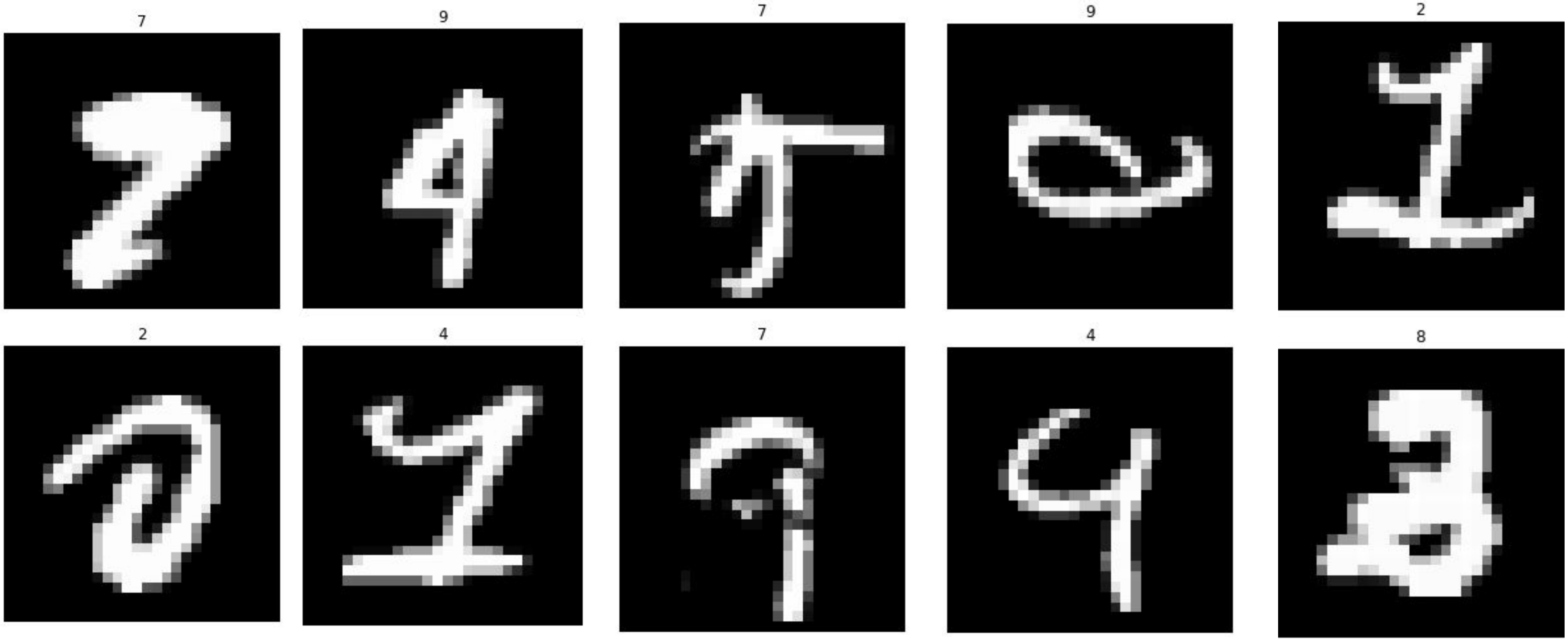
```python
print(X_train[0].shape)
print(len(X_train))
```

```
torch.Size([100, 1, 28, 28])
160
```

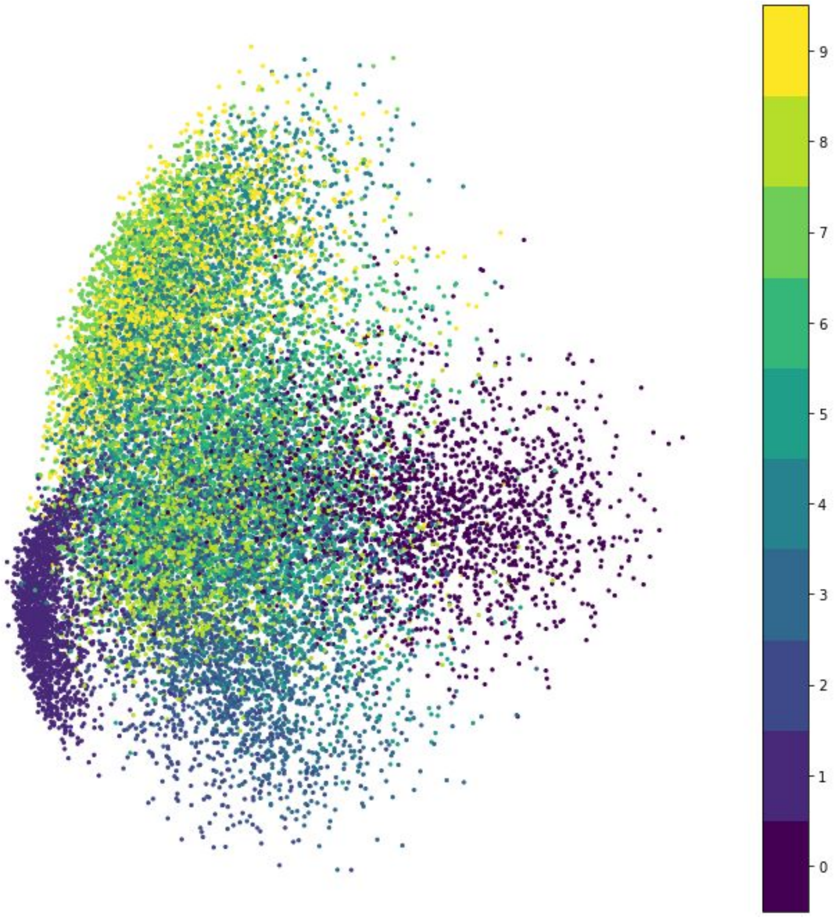
MNIST dataset



MNIST dataset - some wrong predictions



MNIST dataset



Network architecture

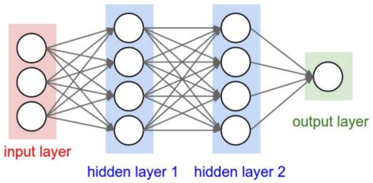
Flow of the information

- **Feedforward network**: single direction towards the output layer and no loops;

- Feedback network: signals can travel in both directions through the loops. Time series and sequential tasks.

Components

- Input Layer;
- Hidden Layers;
- Output Layer;
- Neurons;
- Weights.

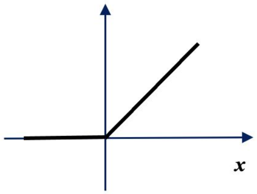


Algorithms

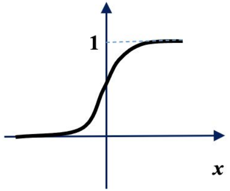
- Training data set: is used for tuning the weights parameters of the NN.

- Test data set: used to check the performance of the NN.

Activation function



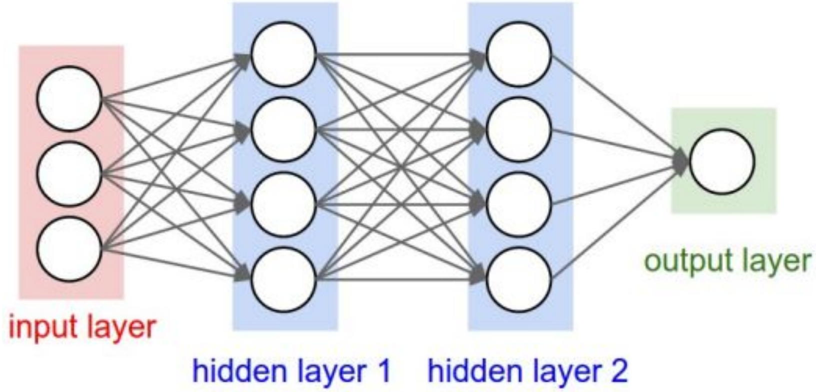
$$\mathrm{Re}\,Lu(x) = \begin{cases} x \\ 0, \end{cases}$$



$$Sigmoid(x) = \frac{1}{1 - e^{-\lambda x}}$$

$$\mathrm{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

Feed-Forward Neural Network



source : brilliant.org

- 4 layers;
- Input layer : 784 neurons;
- Hidden layer 1: 1000 neurons;
- Hidden layer 2: 100
- Output layer: 10

Accuracy on:

- Training data set: 99.6%
- Test data set: 96.6%

Network architecture

```
# Defining the Neural Network
from torch import nn

#Layer details for the neural network
input_size = 28*28
hidden_sizes = [1000, 100]
output_size = 10

#Construct the Sequential Function
#ReLu is activation function
model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[1],output_size),
                      nn.LogSoftmax(dim=1))

print(model)
```

2nd layer (1st hidden layer)

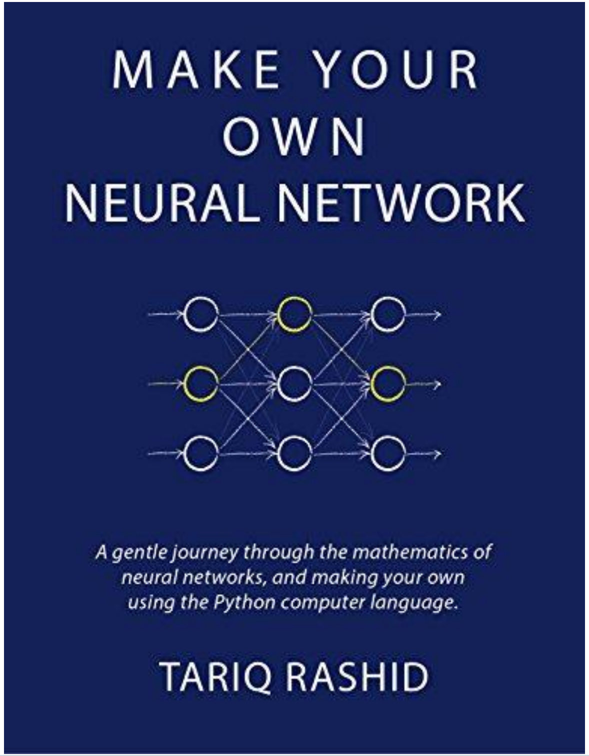
3nd layer (2st hidden layer)

4nd layer (output layer)

Easy example of a pure Numpy network

https://makeyourownneuralnetwork.blogspot.com/

https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork



Easy example of a pure Numpy network

- **Error**: Loss gradient with respect to the ith weighted input

$$\mathbf{e}_i^L = \frac{\partial c(\mathbf{x},\mathbf{y},\theta)}{\partial z_i^L} = \underbrace{\frac{\partial c(\mathbf{x},\mathbf{y},\theta)}{\partial \hat{y}_i}}_{\text{loss derivative}} \frac{\partial \hat{y}_i}{\partial z_i^L} = \frac{\partial c(\mathbf{x},\mathbf{y},\theta)}{\partial \sigma(z_i^L)} \quad \underbrace{\sigma'(z_i^L)}_{\text{activation derivative}}$$

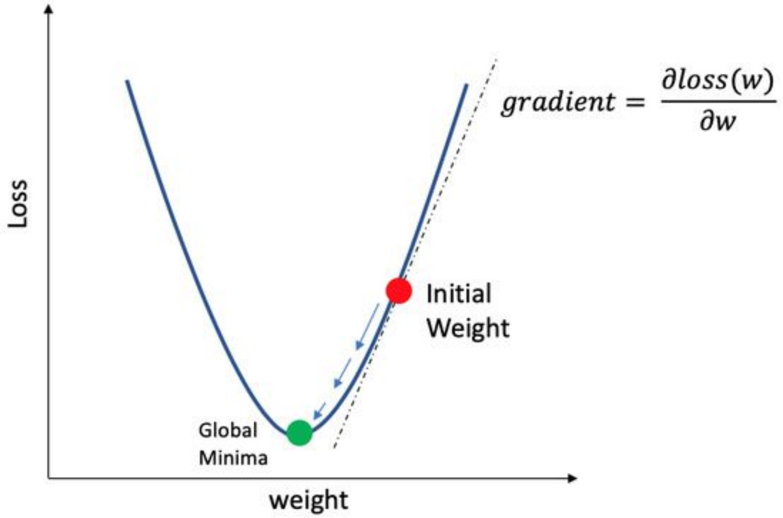
- **Weight update**: $w_{ij}^l \leftarrow w_{ij}^l - \eta \mathbf{e}_i^l x_j^{l-1}$

```
#error is the (target - actual)
output_errors = targets - final_outputs

# hidden layer error = output_errors, split by weights, recombined at hidden nodes
hidden_errors = numpy.dot(self.who.T, output_errors)

# update the weights for the links between the hidden and output layers
self.who += self.lr*numpy.dot(output_errors*final_outputs*(1.0-final_outputs), numpy.transpose(hidden_outputs))
```

a

Loss Functions

- Quantify how good or bad the model is performing.
- Measuring the expected and predicted value.



Loss Function for hot one encoding

One-hot is a group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0).

-Normalize the output.

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

Last Layer

NLLLOSS

The negative log likelihood loss. It is useful to train a classification problem with C classes.

torch.nn.NLLLoss()

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n,y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$$

where x is the input, y is the target, w is the weight, and N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

CrossEntropyLoss

Useful when training a classification problem with C classes

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

torch.nn.CrossEntropyLoss()

How to train your network

- Loss function
- Optimizer
- Backpropagation

Loss

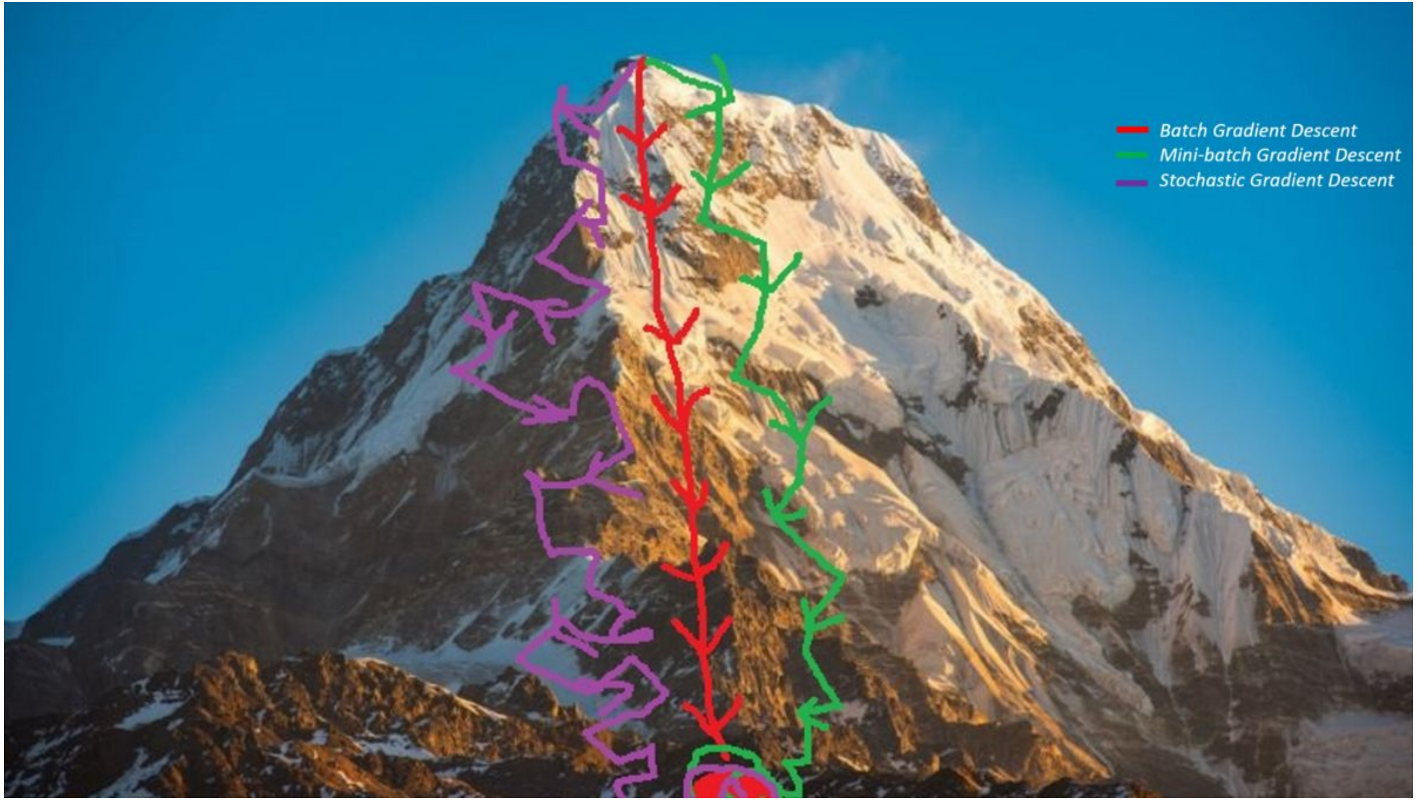
- How close are the predictions to the labels?
- Minimize the loss
- Regression
  - Mean squared errors (nn.MSELoss)
- Classification
  - Negative log likelihood (nn.NLLLoss)
  - Cross entropy loss (nn.CrossEntropyLoss)
    - nn.LogSoftmax + nn.NLLLoss

Optimizer

- How to find the global minimum (of the loss)?

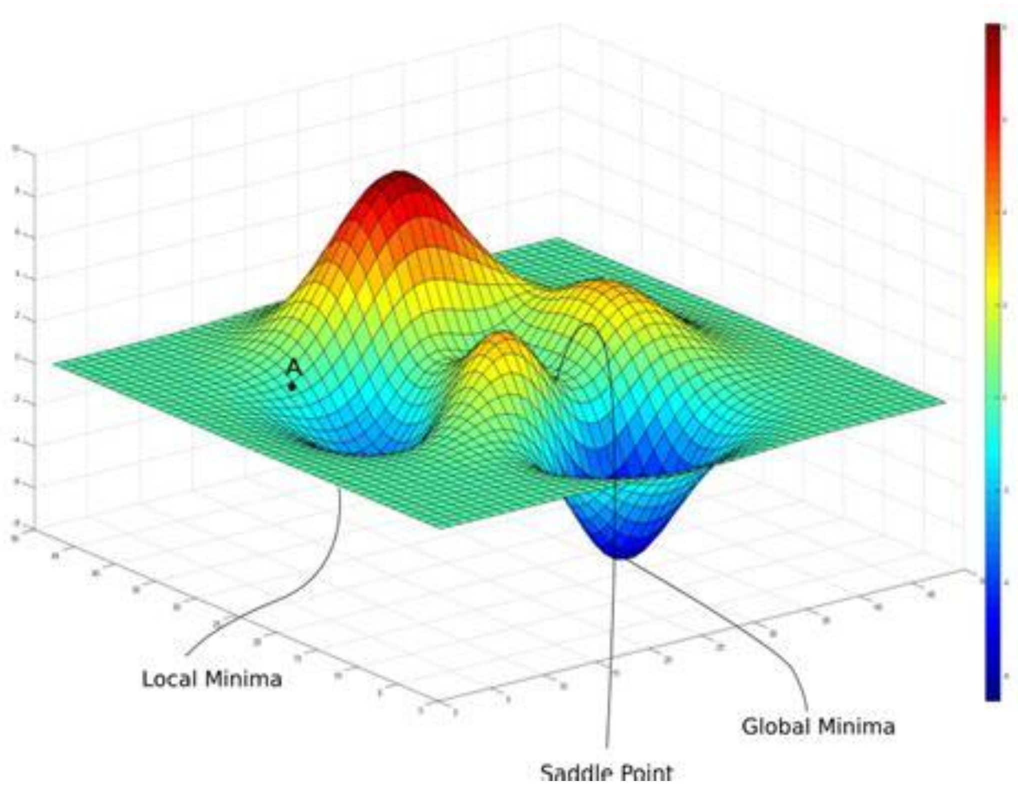
Optimizer

https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3



Optimizer

https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/



Optimizer

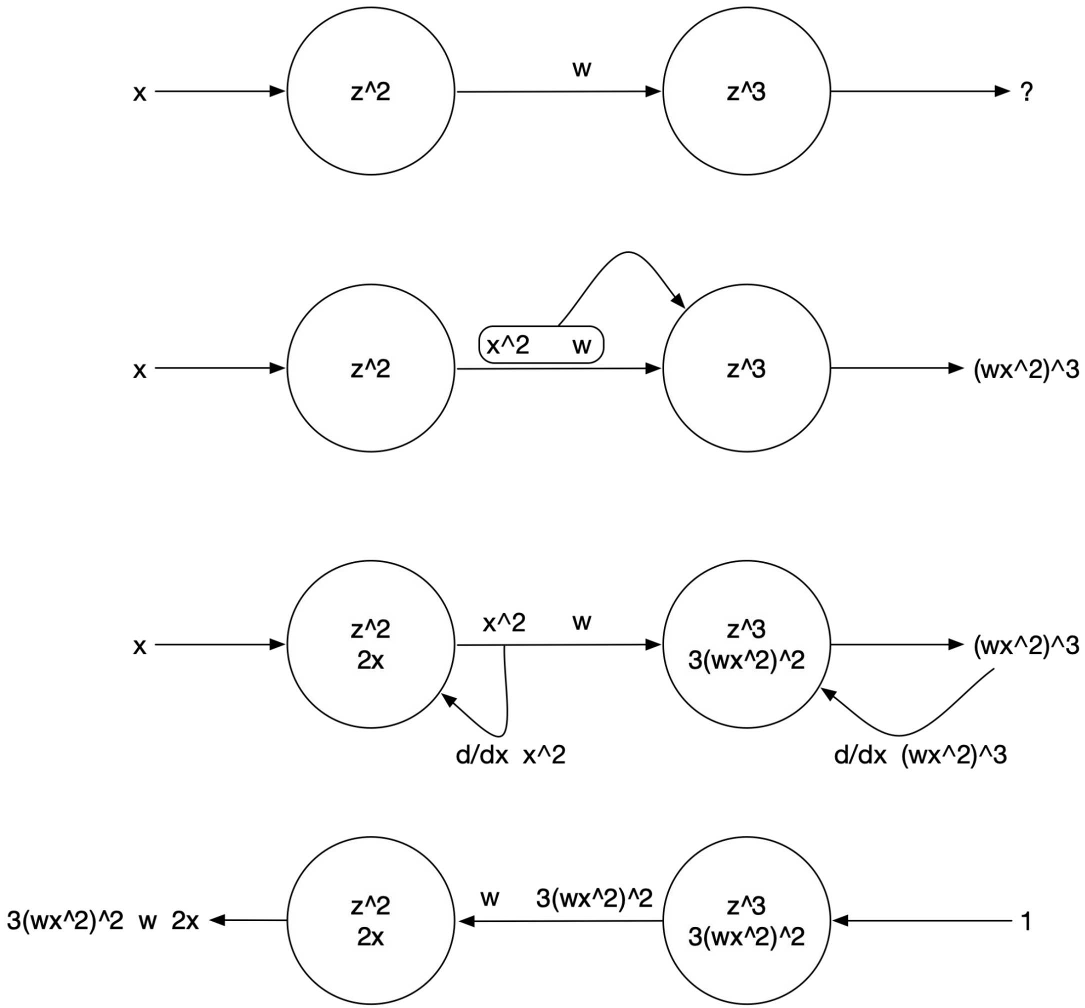
- Typically gradient descent (or variant)
- RMSProp, Adam
  - Dynamic step size depending on momentum
  - Running gradient’s average
  - Running average of 1st (and 2nd) momentum
    - Curvature
  - Speed up convergence by larger LR/steps in flat directions
- PyTorch: `optim = torch.optim.Adam(params=model.parameters(), lr=1e-3)`

Backpropagation

- Gradient computation of the loss function w.r.t. θ
- Automatic differentiation using the chain rule
- Differentiable loss and activation functions

1. In FF pass, store computed values in each neuron
2. Use loss to update weights backwards through layers

Backpropagation - Basic idea



Backpropagation

- Append loss function E to output layer
- Partial derivatives w.r.t. weights
- Add parallel paths in the graph

- Example for single weight:
  - grad = dE/dw_i
  - w_i = w_i - LR*grad

Code

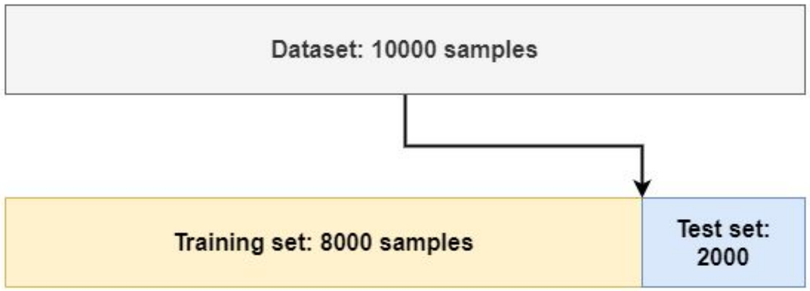
https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

Validation II

- Evaluate the performance of the model.

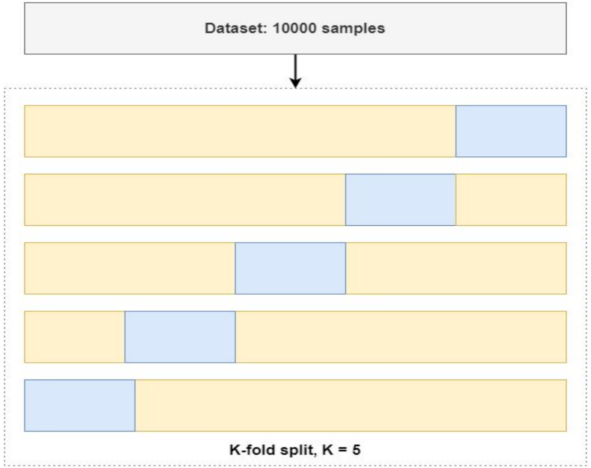
- Simple hold-out split:




```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( data_x, data_y, test_size=0.2)
```

Validation II

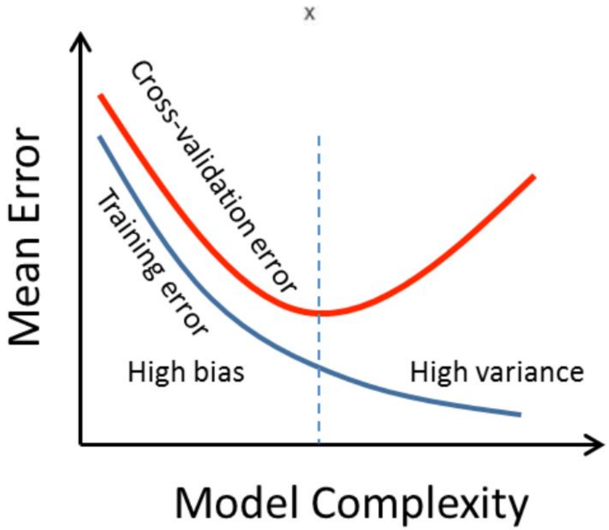
- Cross Validation:




```python
from sklearn.model_selection import KFold

splitter = KFold(n_splits=5)
for train_index, test_index in splitter.split(data_x,data_y):
  train_x = data_x[train_index]
  train_y = data_y[train_index]

  test_x = data_x[test_index]
  test_y = data_y[test_index]

  print(train_x.shape , test_x.shape)
```

Hyperparameter optimization

Hyperparameter selection: Hyperparameters are parameters that cannot not be obtained from the learning algorithm (here LLS).
Example: The type of function ϕ used for training cannot be determined by minimizing the training error.



Hyperparameter optimization

Network architecture

- Number of layers
- Number of neurons per layer
- Type of activation functions

Optimizer

- Learning rate, momentum
- Advanced optimizers e.g. ADAM -> more parameters

Freie Universität Berlin

Hyperparameter optimization

```
input_size = 784
hidden_sizes = [128, 64, 32]
output_size = 10

model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[1], hidden_sizes[2]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[2], output_size),
                      nn.LogSoftmax(dim=1))
```

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Hyperparameter optimization

```
- size of layers:
    200, 100 training: 99 test: 95
    200, 64 training: 99 test: 95
    128, 64 training: 99 test: 92-94
    64, 30 training: 98-99 test: 93-94
    30, 64 training: 97-98 test: 92-93
    30, 30 training: 98 test: 92

- learning rate: using size of layers 64, 30
    0.02 training: 98-99 test: 93-94
    0.01 training: 96-97 test: 93
    0.01 momentum=0.9 training: 99,99 test: 94
    0.005 momentum=0.9 training: 99.9 test: 95
```