Contents

1 Machine Learning Basics

1.1 Un-/Supervised Learning

- Unsupervised Learning
  - No label or target class
  - Find out properties of the structure of the data
  - clustering (k-means, PCA, etc.)
- Supervised Learning
  - Labels or target classes
- Reinforcement Learning

2 Linear Models

2.1 Regression, Classification

- **Regression:** Predicts a continuous output value
- **Classification:** Predicts a discrete value
  - **Binary Classification:** Output either 0 or 1
  - **Multi-class classification:** Set of N classes

2.2 Obtaining the model

1. Estimating using current model
2. Calculating loss
3. Optimizing the model

2.3 Linear Regression

2.3.1 Linear Model

i: index of current sample
j: index of current weight
d: input dimension/number of weights
x_{ij}: i-th input data/feature of the j-th weight
θ_0: bias
θ_j: weights
$\hat{y}_i$: i-th Prediction/Estimation (predicted label)

$$\hat{y}_i = \theta_0 + \sum_{j=1}^{d} x_{ij}\theta_j = \theta_0 + x_{i1}\theta_1 + \cdots + x_{id}\theta_d$$

Matrix Notation:

$$\hat{y} = X\theta$$

2.3.2 Loss function

Measures how good my estimation is and tells the optimization method how to make it better

2.3.2.1 Linear Least Squares:

n: number of training samples
y: Ground truth labels
$\hat{y}$: Estimated labels

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Matrix Notation:

$$J(\theta) = (X\theta - y)^T(X\theta - y) = (\hat{y} - y)^T(\hat{y} - y)$$

2.3.3 Optimization

Changes the model in order to improve the loss function/estimation

$$\theta = (X^T X)^{-1} X^T y$$

2.4 Logistic Regression

2.4.1 Model

i: index of current sample
j: index of current weight
d: input dimension/number of weights
x_{ij}: i-th input data/feature of the j-th weight
θ_j: model parameters
$\hat{y}_i$: i-th Prediction/Estimation (predicted label)

$$\hat{y}_i = \sigma(x_i\theta) = \sigma\left(\sum_{j=1}^{d} x_{ij}\theta_j\right)$$

with

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

2.4.2 Loss function

Measures how good my estimation is and tells the optimization method how to make it better

2.4.2.1 Binary Cross-Entropy:

y: Ground truth labels
$\hat{y}$: Estimated labels

$$\mathcal{L}(\hat{y}_i, y_i) = y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)$$

2.4.3 Cost function

n: number of labels

$$C(\theta) = -\frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\hat{y}_i, y_i)$$

2.4.4 Optimization

Changes the model in order to improve the loss function/estimation

- No closed-form solution
- Make use of an iterative method e.g. Gradient Descent

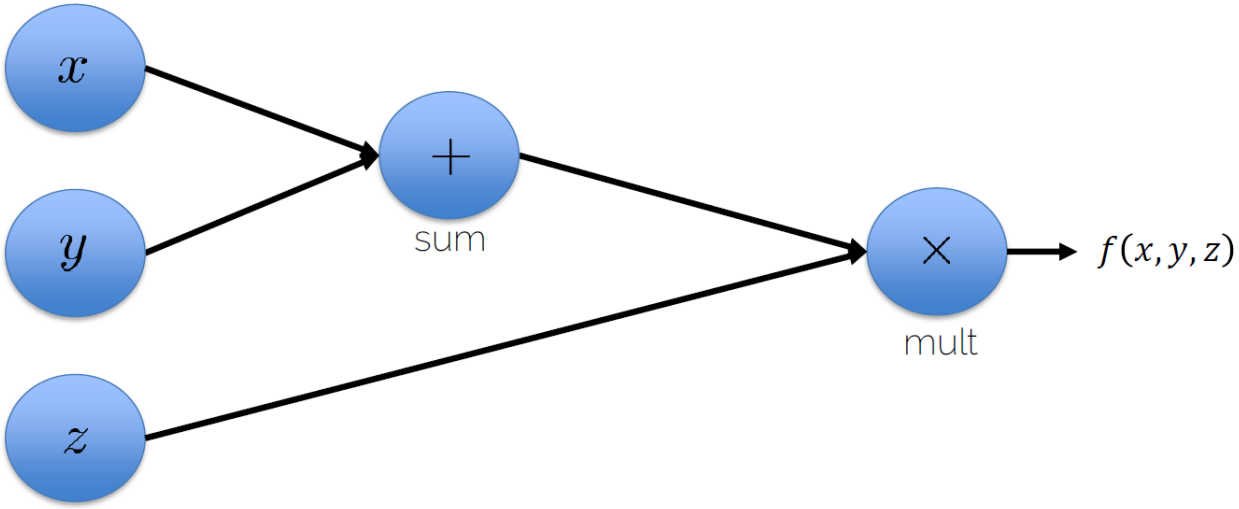
3 Computational Graphs

- Directional graph
- Matrix operations are represented as compute graphs
- Vertex nodes are variables or operators like $+, -, *, /, log(), exp(), \ldots$
- Directional edges show flow of inputs to vertices
- Neural network can be represented as computational graph

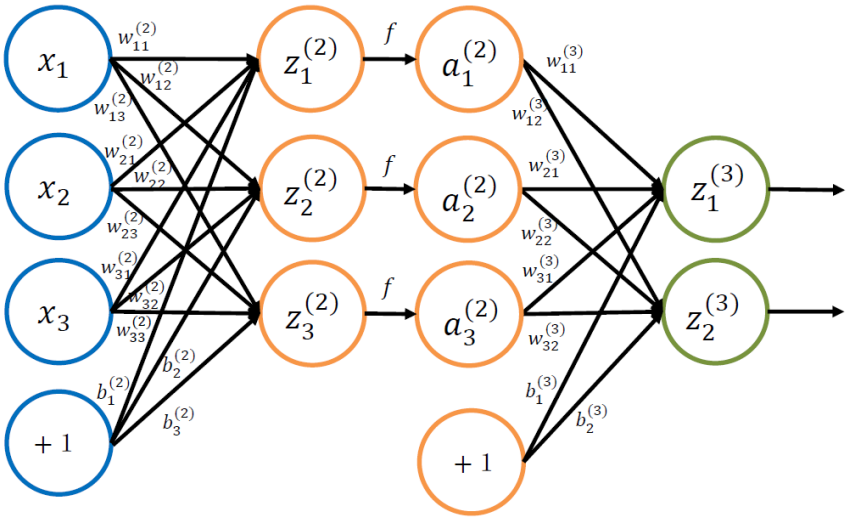
3.1 Graphical representation

Example

$$f(x, y, z) = (x + y) \cdot z$$



Neural Network as Computational Graph



$$z_k^{(2)} = \sum_i x_i w_{ik}^{(2)} + b_i^{(2)}$$

$$a_k^{(2)} = f(z_k^{(2)})$$

$$z_k^{(3)} = \sum_i a_i^{(2)} w_{ik}^{(3)} + b_i^{(3)}$$

...

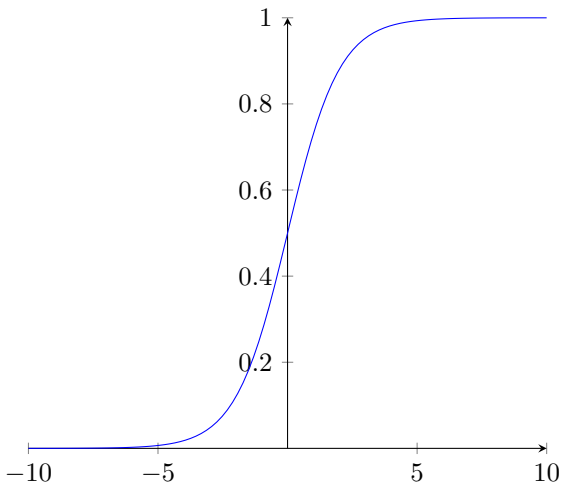
4 Neural Network

4.1 Activation Functions

4.1.1 Description

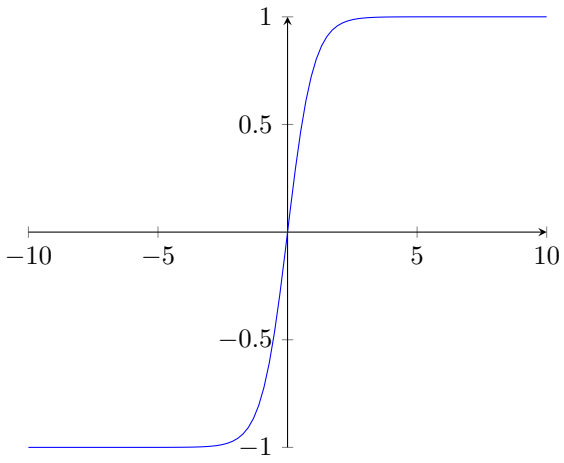
4.1.2 Sigmoid

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



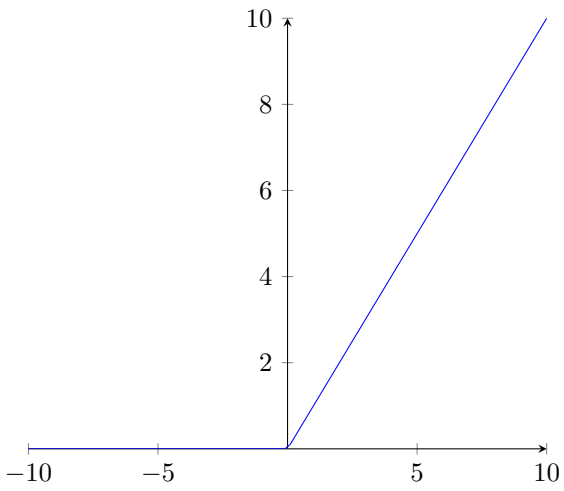
4.1.3 Tanh(x)

$$\tanh(x)$$



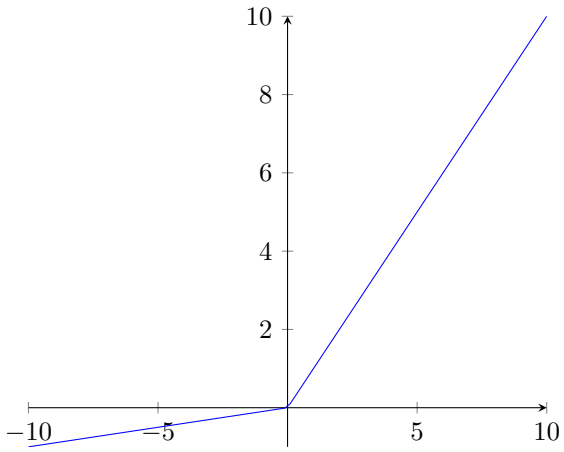
4.1.4 ReLU

$$\max(0, x)$$



4.1.5 Leaky ReLU

$$\max(0.1x, x)$$



4.1.6 Parametric ReLU

$$\max(\alpha x, x)$$

4.1.7 ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

4.1.8 Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

4.2 Loss Function

4.2.1 Description

A function to measure the goodness of the predictions

- Large loss $\implies$ bad predictions
- Goal: Minimize the loss $\iff$ Find better predictions
- Choice of the loss function depends on the concrete problem or the distribution of the target variable

4.2.2 Parameters

y: Ground truth
$\hat{y}$: Prediction
n: number of training samples
k: number of classes

4.2.3 L1 Loss

$$\mathcal{L}(y, \hat{y}; \theta) = \frac{1}{n} \sum_{i}^{n} ||y_i - \hat{y}_i||_1$$

4.2.4 MSE Loss

$$\mathcal{L}(y, \hat{y}; \theta) = \frac{1}{n} \sum_{i}^{n} ||y_i - \hat{y}_i||_2^2$$

4.2.5 Binary Cross Entropy

$$\mathcal{L}(y, \hat{y}; \theta) = -\sum_{i=1}^{n} (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))$$

4.2.6 Cross Entropy

$$\mathcal{L}(y, \hat{y}; \theta) = -\sum_{i=1}^{n} \sum_{j=1}^{k} (y_{ij} \log \hat{y}_{ij})$$

4.3 Optimization Functions

4.3.1 General Optimization

- **Goal:** $\theta^* = \arg\min f(\theta, x, y)$
- **Linear Systems (Ax = b)**
  - LU, QU, Cholesky, Jacobi, Gauss-Seidel, CG, PCG, ...
- **Non-linear systems** (Gradient based methods):
  - First order methods:
    - Gradient Descent, SGD, SGD with Momemtum, RMSProp, Adam (Standard)
  - Second order methods (faster than first order methods, but only for full batch updates):
    - Newton, Gauss-Newton, Levenberg-Marquardt, (L)BFGS

4.3.2 Gradient Descent

- Finds local minimum
- Does not guarantee to find global optimum
- Does gradient steps in direction of negative gradient
- Requires a lot of memory $\implies$ extremely expensive

Parameters:

$f(\theta, x_{1..n}, y_{1..n})$:	Function describing the neural network (including loss function)
$x_{1..n}$:	Input vectors for all n training samples
$y_{1..n}$:	Ground truth for all n training samples
$\theta^k = \{W, b\}$:	Model Parameters at step k
α:	Learning rate

Gradient Step:

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta f(\theta^k, x_{1..n}, y_{1..n})$$

4.3.3 Stochastic Gradient Descent

- Split training set into several minibatches
- Minibatch size:
  - Is a hyperparameter
  - Is typically a power of 2
  - Smaller batch size $\implies$ Greater variance in the gradients
  - Is mostly limited by GPU memory
- Epoch: Complete pass through training set
- Cannot independently scale directions
- Need to have conservative min learning rate to avoid divergence
- Is slower than necessary

Parameters:

n:	Number of total training samples
m:	Minibatch size (number of training samples per minibatch)
n/m:	Number of minibatches
$f(\theta, x_{1..m}, y_{1..m})$:	Function describing the neural network (including loss function)
$x_{1..m}$:	Input vectors for one minibatch
$y_{1..m}$:	Ground truth for one minibatch
$\theta^k = \{W, b\}$:	Model Parameters at step k
k:	iteration in current epoch
α:	Learning rate

Gradient Step:

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta f(\theta^k, x_{1..m}, y_{1..m})$$

4.3.3.1 Convergence of Stochastic Gradient Descent

$f(\theta, x, y)$ converges to a local (global) minimum if:

1. $\alpha_n \geq 0, \forall n \geq 0$
2. $\sum_{n=1}^{\infty} \alpha_n = \infty$
3. $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$
4. $f(\theta, x, y)$ ist strictly convex

where $\alpha_1, \ldots, \alpha_n$ is a sequence of positive step-sizes

4.3.4 Gradient Descent with Momentum

- Step will be largest when a sequence of gradients all point to the same direction

Parameters:

$f(\theta, x, y)$:	Function describing the neural network (including loss function)
x:	Input vectors
y:	Ground truth
$\theta^k = \{W, b\}$:	Model Parameters at step k
α:	Learning rate
β:	Accumulation rate (friction, momentum), default: 0.9
v^k:	velocity at step k

Gradient step:

$$v^{k+1} = \beta \cdot v^k - \alpha \cdot \nabla_\theta f(\theta^k, x, y)$$
$$\theta^{k+1} = \theta^k + v^{k+1}$$

4.3.5 Nesterov Momentum

- Look-ahead momentum
- Steps:
  1. Make a big jump in the direction of the previous accumulated gradient
  2. Measure the gradient where you end up
  3. Make a correction

Parameters:

$f(\theta, x, y)$:	Function describing the neural network (including loss function)
x:	Input vectors
y:	Ground truth
$\theta^k = \{W, b\}$:	Model Parameters at step k
α:	Learning rate
β:	Accumulation rate (friction, momentum), default: 0.9
v^k:	velocity at step k

Gradient step:

$$\tilde{\theta}^{k+1} = \theta^k + \beta \cdot v^k$$
$$v^{k+1} = \beta \cdot v^k - \alpha \cdot \nabla_\theta f(\tilde{\theta}^{k+1}, x, y)$$
$$\theta^{k+1} = \theta^k + v^{k+1}$$

4.3.6 Root Mean Squared Prop (RMSProp)

- Divides the learning rate by an exponentially-decaying average of squared gradients
- Damps the oscillations for high-variance directions

\+ Can increase learning rate because it is less likely to diverge → Speeds up learning speed

Parameters:

$f(\theta, x, y)$:	Function describing the neural network (including loss function)
x:	Input vectors
y:	Ground truth
$\theta^k = \{W, b\}$:	Model Parameters at step k
α:	Learning rate
β:	Accumulation rate (friction, momentum), default: 0.9
ϵ:	Prevents division by zero, default: 10^{-8}
s^k:	Second momentum (uncentered variance of gradients)

Gradient step:

$$s^{k+1} = \beta \cdot s^k + (1 - \beta)(\nabla_\theta f(\theta^k, x, y) \circ \nabla_\theta f(\theta^k, x, y))$$
$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\nabla_\theta f(\theta^k, x, y)}{\sqrt{s^{k+1}} + \epsilon}$$

where $a \circ b$ is an element-wise multiplication

4.3.7 Adaptive Momemt Esimation (Adam)

- Combines Momentum and RMSProp
- Combines first and second order momentum

Parameters:

$f(\theta, x, y)$:	Function describing the neural network (including loss function)
x:	Input vectors
y:	Ground truth
$\theta^k = \{W, b\}$:	Model Parameters at step k
α:	Learning rate
β_1:	Accumulation rate 1, default: 0.9
β_2:	Accumulation rate 2, default: 0.999
ϵ:	Prevents division by zero, default: 10^{-8}
s^k:	Second momentum (uncentered variance of gradients)

Gradient step:

$$\hat{m}^{k+1} = \frac{\beta_1 \cdot m^k + (1-\beta_1) \cdot \nabla_\theta f(\theta^k, x, y)}{1-\beta_1^{k+1}}$$

$$\hat{v}^{k+1} = \frac{\beta_2 \cdot v^k + (1-\beta_2)(\nabla_\theta f(\theta^k, x, y) \circ \nabla_\theta f(\theta^k, x, y))}{1-\beta_2^{k+1}}$$

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\hat{m}^{k+1}}{\sqrt{\hat{v}^{k+1}} + \epsilon}$$

where $m^0 = v^0 = 0$

4.3.8 Newton's Method

- Computation complexity of inversion per iteration: $\mathcal{O}(k^3)$

Parameters:

$f(\theta)$:	Function describing the neural network (including loss function)
$\theta = \{W, b\}$:	Model Parameters
$\nabla_\theta f(\theta)$:	Gradient (first derivative)
$H(\theta)$:	Hessian matrix (second derivative)

Approximate the function by a second-order Taylor series expansion

$$f(\theta) \approx f(\theta_0) + (\theta - \theta_0)^T \nabla_\theta f(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H (\theta - \theta_0)$$

Update step:

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta f(\theta)$$

4.3.9 Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS and L-BFGS)

- Belongs to the family of quasi-Newton methods
- Have an approximation of the inverse of the Hessian
- Computation complexity of inversion per iteration:
  - BFGS: $\mathcal{O}(n^2)$
  - L-BFGS: $\mathcal{O}(n)$

Update step:

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta f(\theta)$$

4.3.10 Gauss-Newton

- True 2nd derivatives are often hard to obtain $\rightarrow$ Approximate

Parameters:

$\theta = \{W, b\}$:	Model Parameters
$\nabla_\theta f(\theta)$:	Gradient (first derivative)
$\mathcal{J}(\theta)$:	Jacobian matrix

$$H(\theta) \approx 2\mathcal{J}^T(\theta)\mathcal{J}(\theta)$$

Linear equation:

$$2(\mathcal{J}^T(\theta_k)\mathcal{J}(\theta_k)) \cdot (\theta_k - \theta_{k+1}) = \nabla_\theta f(\theta)$$

4.3.11 Levenberg

- Damped version of Gauss-Newton
- Damping factor is adjusted in each iteration, so that: $f(\theta_k) > f(\theta_{k+1})$

Parameters:
$\theta = \{W, b\}$: Model Parameters
$\nabla_\theta f(\theta)$: Gradient (first derivative)
$\mathcal{J}(\theta)$: Jacobian matrix
λ: Damping factor

Linear equation:

$$(\mathcal{J}^T(\theta_k)\mathcal{J}(\theta_k) + \lambda I) \cdot (\theta_k - \theta_{k+1}) = \nabla_\theta f(\theta)$$

4.3.12 Levenberg-Marquardt

- Avoids slow convergence in components with a small gradient

Parameters:
$\theta = \{W, b\}$: Model Parameters
$\nabla_\theta f(\theta)$: Gradient (first derivative)
$\mathcal{J}(\theta)$: Jacobian matrix
λ: Damping factor

Linear equation:

$$(\mathcal{J}^T(\theta_k)\mathcal{J}(\theta_k) + \lambda \cdot \mathrm{diag}(\mathcal{J}^T(\theta_k)\mathcal{J}(\theta_k))) \cdot (\theta_k - \theta_{k+1}) = \nabla_\theta f(\theta)$$

4.4 Learning rate

- Goal: High learning rate in the beginning, then low learning rate

4.4.1 Learning Rate Decay

α_0: Initial learning rate (e.g. 0.1)
t: factor by which the learning rate is decayed
$epoch$: epoch of current run

Learning rate decays:

$$\alpha = \frac{1}{1 + t \cdot epoch} \cdot \alpha_0, \qquad \text{default: } t = 0.1$$

Step decay:

$$\alpha = \alpha - t \cdot \alpha, \qquad \text{default: } t = 0.5$$

Exponential decay:

$$\alpha = t^{epoch} \cdot \alpha_0, \qquad t < 1.0$$

$$\alpha = \frac{t}{\sqrt{epoch}} \cdot \alpha_0$$

4.4.2 Training Schedule

- Manually set learning rate every n epochs

4.5 Regularization Techniques

- Increasing training error
- Lower validation error

4.5.1 L1/L2 Regularization

L:	Loss
$\mathcal{L}(y, \hat{y}, \theta)$:	Loss function (without generalization)
λ:	Regularization rate
$\theta = \{W, b\}$:	Model parameters

Add regularization term to loss function:

$$L = \mathcal{L}(y, \hat{y}, \theta) + \lambda R(\theta)$$

L1 Regularization
Enforces sparsity

$$R(\theta) = \sum_{i=1}^{n} |\theta_i|$$

L2 Regularization
Enforces that the weights have similar values

$$R(\theta) = \sum_{i=1}^{n} \theta_i^2$$

4.5.2 Dropout

4.5.3 Early Stopping

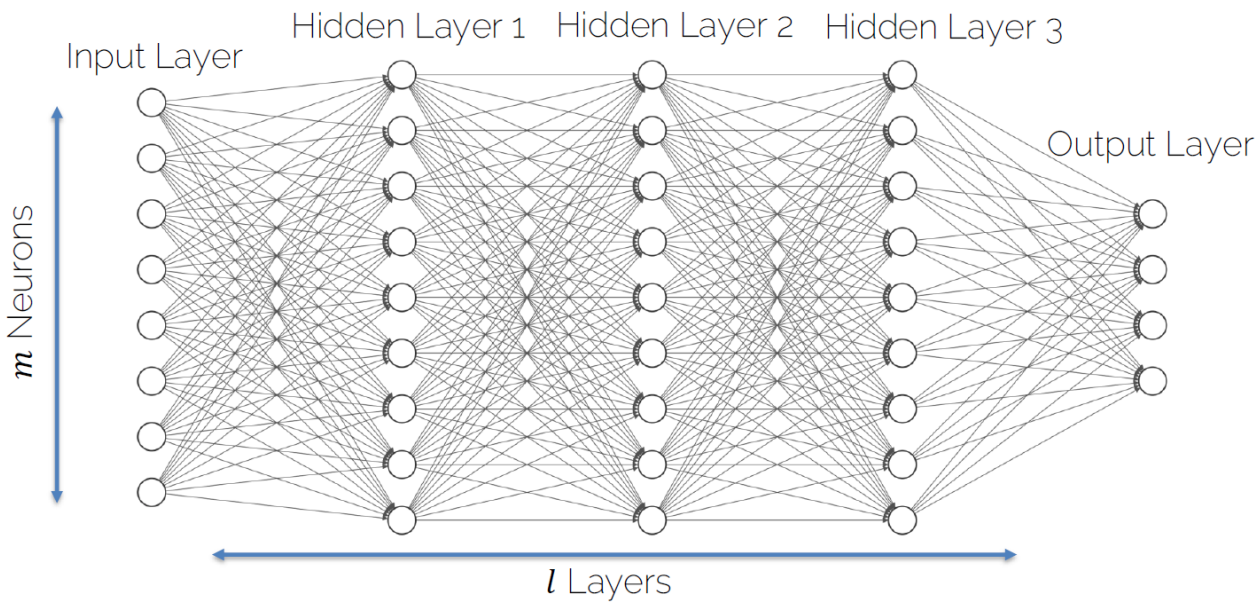
5 Fully Connected Neural Network

5.1 Structure

Parameters:

x_k:	Input variables
$\theta = \{W, b\}$:	Model parameters
$w_{i,j,k}$:	Network weights
$b_{i,j}$:	Network biases
	i: Index of layer
	j: Index of neuron in layer (neuron of next layer)
	k: Index of weight in neuron (neuron of previous layer)
l:	Depth: number of layers (All hidden and the output layer - no input layer)
m:	Width: number of neurons in layer (Can be different for each layer)
n:	Number of weights in neuron
$\hat{y}_i$:	Computed output
y_i:	Ground truth targets
$\mathcal{L}(y, \hat{y}, \theta)$:	Loss function

Graphical Representation



Mathematical Representation

$$L = \mathcal{L}(y_j, \hat{y}_j, \theta)$$
$$\hat{y}_j = a(s_{l,j})$$
$$h_{i,j} = a(s_{i,j})$$
$$s_{i,j} = b_{i,j} + \sum_{k=1}^{m} h_{i-1,k} \cdot w_{i,j,k}$$
$$s_{1,j} = b_{1,j} + \sum_{k=1}^{m} x_k \cdot w_{1,j,k}$$

5.2 Number of weights

l: Depth: number of layers
m_i: Width: number of neurons in layer i (Here: layer 0 is the input layer)

Number of weights is defined as:

$$\sum_{i=1}^{l} m_i \cdot m_{i-1} + m_i$$

5.3 Forward and Backward Pass

5.3.1 Forward Pass/ Forward Propagation

Use formulas to calculate loss:

$$s_{1,1} = b_{1,1} + \sum_{k=1}^{m} x_k \cdot w_{1,1,k} \qquad \ldots \qquad L = \mathcal{L}(y_j, \hat{y}_j)$$

5.4 Backward Pass/Backward Propagation

Weights of last layer:

$$\frac{\partial L}{\partial w_{l,j,k}} = \frac{\partial L}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial s_{l,j}} \cdot \frac{\partial s_{l,j}}{\partial w_{l,j,k}}$$

Weights of second last layer:

$$\frac{\partial L}{\partial w_{l-1,j,k}} = \sum_{o=1}^{m} \frac{\partial L}{\partial \hat{y}_o} \cdot \frac{\partial \hat{y}_o}{\partial s_{l,o}} \cdot \frac{\partial s_{l,o}}{\partial h_{l-1,j}} \cdot \frac{\partial h_{l-1,j}}{\partial s_{l-1,j}} \cdot \frac{\partial s_{l-1,j}}{\partial w_{l-1,j,k}}$$

General:

$$\frac{\partial L}{\partial w_{i,j,k}} = \sum_{o_1=1}^{m} \cdots \sum_{o_p=1}^{m} \frac{\partial L}{\partial \hat{y}_{o_1}} \cdot \frac{\partial \hat{y}_{o_1}}{\partial s_{l,o_1}} \cdot \frac{\partial s_{l,o_1}}{\partial h_{l-1,o_2}} \cdot \frac{\partial h_{l-1,o_2}}{\partial s_{l-1,o_2}} \cdot \ldots \cdot \frac{\partial s_{i+1,o_p}}{\partial h_{i,j}} \cdot \frac{\partial h_{i,j}}{\partial s_{i,j}} \cdot \frac{\partial s_{i,j}}{\partial w_{i,j,k}}$$

6 Training

6.1 Learning

- Learning means generalization to unknown dataset, i.e. train on known dataset, test with optimized parameters on unknown dataset

6.2 Dataset

- Split dataset into
  - Training data (e.g. 60%, 80%) - Used to train the model
  - Validation data (e.g. 20%, 10%) - Validate the current model to find the best hyperparameters
  - Test data (e.g. 20%, 10%) - Is only used once in the end

6.3 Errors

- Ground truth error
  - Faults in dataset, e.g. wrong classification of sample image
  - Underfitting
- Training set error
  - Underfitting
- Validation/test set error
  - Overfitting

6.4 Hyperparameters

- Hyperparameters = Learning Setup + Optimization, i.e.,
  - Network architecture (number of layers, number of weights, ...)
  - Number of iterations
  - Learning rate(s)
  - Regularization
  - Batch size
  - ...

6.4.1 Hyperparameter Tuning Methods

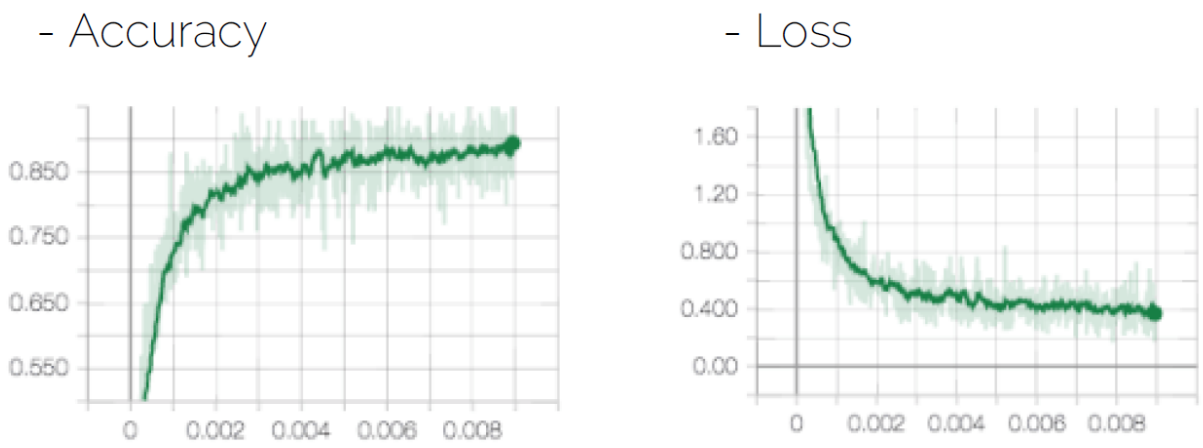
- Manual search:
  - Find out the optimal hyperparameters manually
  - most common
- Grid search:
  - Define ranges for all parameters spaces and select points
  - Iterates over all possible configurations
- Random search:
  - Like grid search but one picks points at random in the predefined ranges

6.5 Learning Curves

6.5.1 Ideal Training

- Small gap between training and validation loss
- Training and validation loss go down at the same rate (stable without fluctuations)

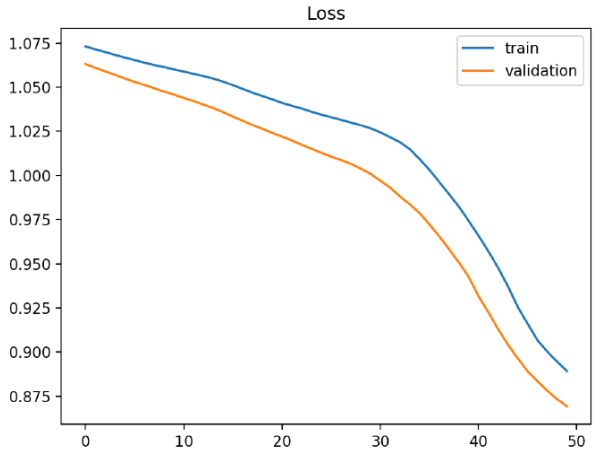
Example:



6.5.2 Underfitting

- Training and validation losses decreases even at the end of training
- Reasons:
  - Model is still learning

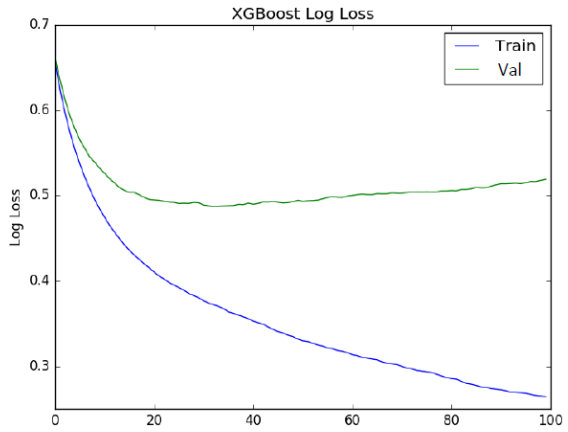
Example:

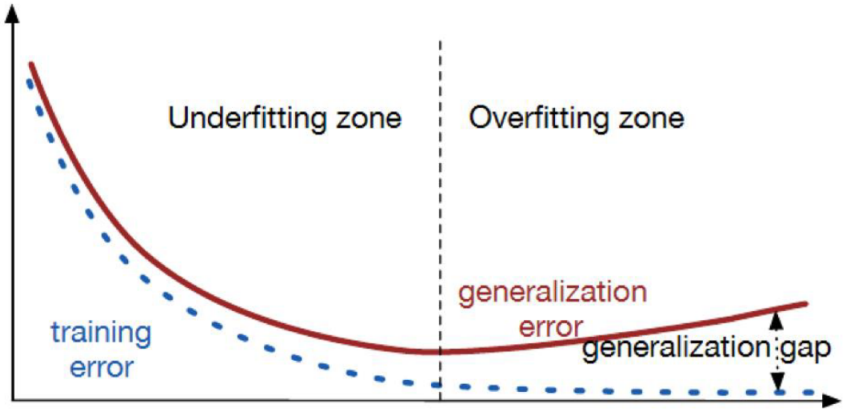


6.5.3 Overfitting

- Training loss decreases and validation loss increases
- Reasons:
  - Model is memorizing the training samples instead of generalizing

Example:



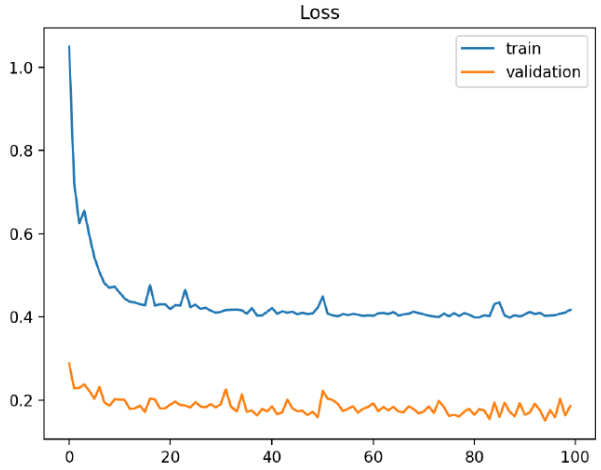


6.5.4 Other Examples

Validation set easier than training set:

- Validation loss is lower than training loss
- Reasons:
  - Validation set is easier than the training set
  - Bug in the implementation

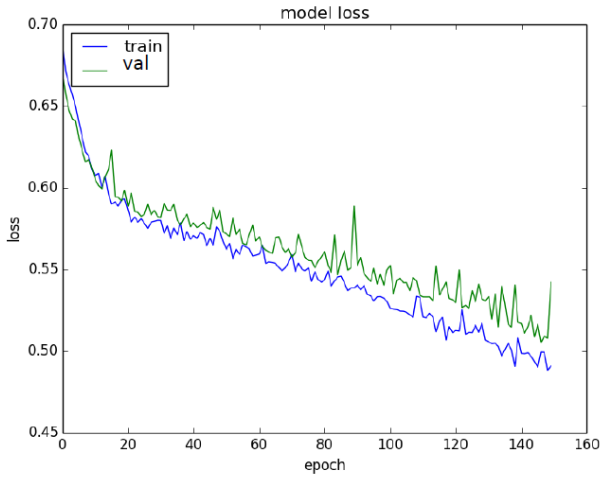
Example:



Learning rate to low:

- Loss curves decrease almost linearly
- Reasons:
  - The initial Learning rate is too low

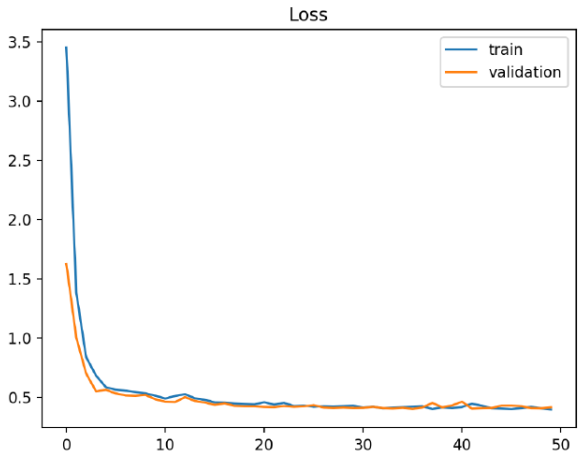
Example:



Learning rate to high:

- Loss curves decrease very quickly at the beginning and then remain on plateaus
- Reasons:
  - Learning rate is too big
  - Inconsistent dataset

Example:



6.6 How To

6.6.1 Network Architecture

- Start with the simplest network possible

6.6.2 Training samples

1. Start with a single training sample
   - Check if output is correct
   - Should overfit
   - Train accuracy should be 100%
2. Increase to handful of samples (e.g., 4)
   - Check if input is handled correctly
3. Move to more samples
   - 5, 10, 100, 1000, ...
   - At some point, you should see generalization

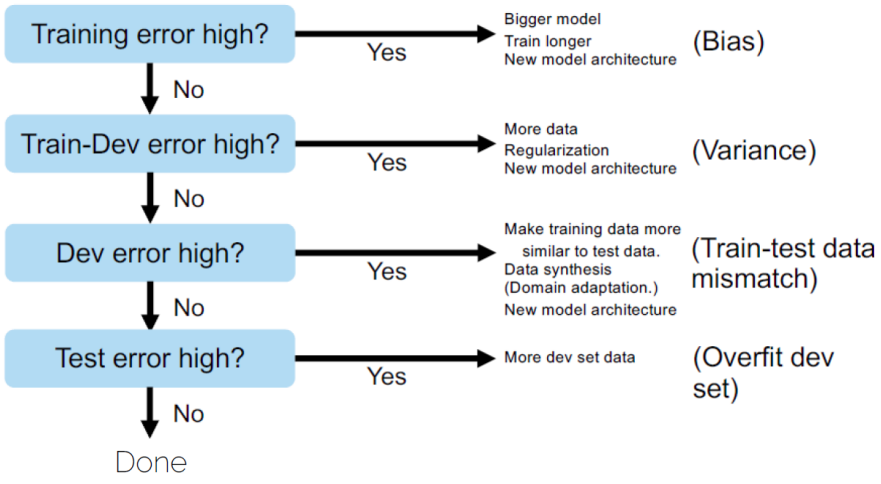
6.6.3 Learning rate

- Find a learning rate that makes the loss drop significantly within 100 iterations
- Good learning rates to try: $1e-1, 1e-2, 1e-3, 1e-4$
- Good weight decay to try: $1e-4, 1e-5, 0$
- Use Grid/Random search

6.6.4 Timings

- Measure how long each iteration takes (should be $< 500ms$
- Look for bottlenecks (e.g. Dataloading, Backpropagation)
- Estimate total time

6.6.5 Basic Recipe



6.6.6 Bad Signs

- Training error not going down
- Validation error not going down
- Performance on validation better than on training set
- Tests on train set different than during training

6.6.7 Good/Bad Practice

Good Practice:

- Use train/validation/test curves
  - Evaluation needs to be consistent
  - Numbers need to be comparable
- Only make one change at a time

Bad Practice/Common Mistakes

- Using single batch it did not overfit
- Forgot to toggle train/eval mode for network
- Forgot to call `.zero_grad()` (in PyTorch) before calling `.backward()`
- Passed softmaxed outputs to a loss function that expects raw logits
- Training set contains test data
- Debug algorithm on test data

Notes

This is a summary of the lecture Introduction to Deep Learning of the Technical University Munich. This lecture was presented by Nießner M. in the summer semester 2020. This summary was created by Gaida B. All provided information is without guarantee.