LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Deep Learning for Physicists

Lecture #4: Fully-connected neural networks

Kosmas Kepesidis

Outline

- Summary of terms so far
- Fully-connected neural networks
  - Fully-connected network with N layers
  - Regression example: earthquake epicenter
  - Classification example: image classification
  - Challenges of training
  - Residuals learning
  - Batch normalization
  - Self-normalization

Summary of terms so far

Summary of terms so far

- **Neural networks** are **models** representing functions/mappings
- Network parameters are adapted to serve specific tasks
- Input variables are usually referred to as **input features**
- The **output** of a network is a **prediction**
- Tasks neural networks are used for, include **classification** (discrete target) and **regression** (continuous target)

Summary of terms so far

- Building blocks of neural networks: **layers** of **nodes**
- At each node, two operations are performed: **affine mapping** and **nonlinear activation function**
- The type and morphology of a network is summarized by the so-called **hyperparameters**
- More abstract or complex features are generated by hidden layers – **feature hierarchy**

Summary of terms so far

- **Learning** is the procedure that leads to a neural network, capable of performing a particular task
- **Training** is the tuning of the network's parameters (**W**, $\vec{b}$) by minimizing an **objective function**
- Individual adaptation of each parameter is achieved by gradients $\frac{\partial \mathcal{L}}{\partial W_i}$ through **backpropagation** and **stochastic gradient descent**

Summary of terms so far

- **Data** in physics could be of various types: sequences of measurements, images, vectors, heterogeneous variable sets
- Data should be appropriately **preprocessed** (scaling, normalization, etc.), before inserted into a neural network
- **Labeled data** include a **target variable** (label) along with the input data: **supervised learning**

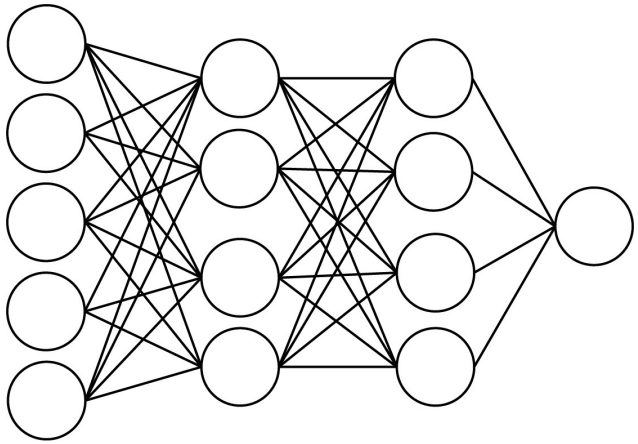
Summary of terms so far

- For obtaining statistically-meaningful results, the networks predictions are validated and tested on unseen data
- The data set is typically split into three sets: **train**, **validation** and **test** sets
- The **train set** is used multiple times during optimization: **epochs**
- Each training step is done on sub-samples of data called **minibatches** and the performance is measured on the **validation set**
- Final statistical evaluation of networks performance is done on the left-out **test set**

Fully-connected neural networks

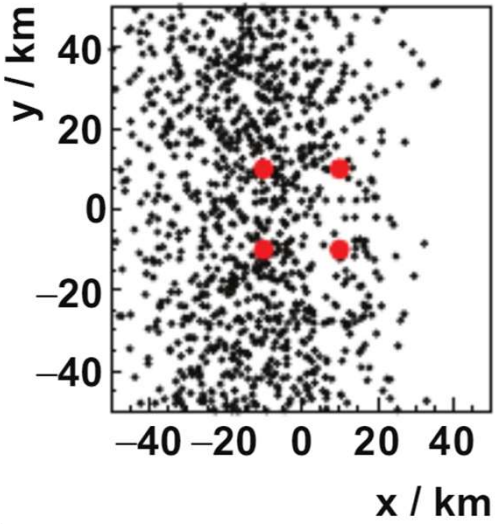
Fully-connected network with N layers

- Traditional neural networks consist of fully-connected layers
- By adding many layers consecutively, the power of **deep learning** can be exploited
- We start our investigation of deep learning methods with two examples: a regression and a classification task
  - M. Erdmann et al., Deep Learning for Physics Research (2021)



Regression example: earthquake epicenter

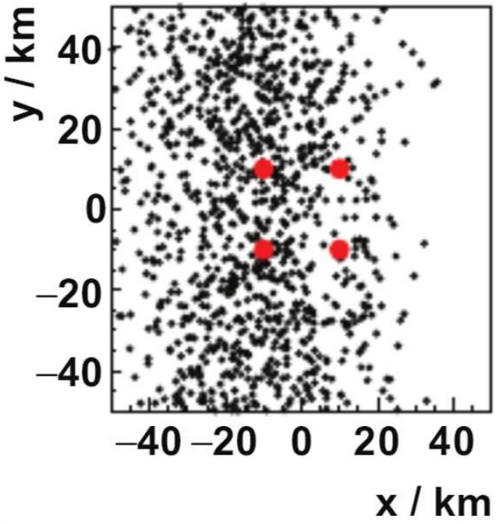
- Task: teach a network to identify the location of the epicenter of earthquakes
- A data set of 1000 simulated earthquakes (black dots) is used, while the arrival time of compression waves t_p and shear waves is recorded by four hypothetical stations (red dots)



i	x/km	y/km	$t_p^{(1)}$/s	$t_s^{(1)}$/s	$t_p^{(2)}$/s	$t_s^{(2)}$/s	$t_p^{(3)}$/s	$t_s^{(3)}$/s	$t_p^{(4)}$/s	$t_s^{(4)}$/s
1	-15.5	-34.6	6.52	9.00	7.25	10.32	3.39	4.98	5.06	7.06
2	3.8	-7.7	3.31	4.51	2.59	3.79	1.79	2.75	0.95	1.30
3	-0.2	29.1	3.17	4.31	3.00	4.37	5.54	8.00	5.77	8.06
4	-16.4	-25.4	5.23	7.21	6.22	8.87	2.17	3.27	4.37	6.09
...	...									

Regression example: earthquake epicenter

- Labels: earthquake locations ($\hat{x}$, $\hat{y}$)
- Input data: information on arrival times of two types of waves for each station
- Network should learn to predict locations based on arrival times



i	x/km	y/km	$t_p^{(1)}$/s	$t_s^{(1)}$/s	$t_p^{(2)}$/s	$t_s^{(2)}$/s	$t_p^{(3)}$/s	$t_s^{(3)}$/s	$t_p^{(4)}$/s	$t_s^{(4)}$/s
1	-15.5	-34.6	6.52	9.00	7.25	10.32	3.39	4.98	5.06	7.06
2	3.8	-7.7	3.31	4.51	2.59	3.79	1.79	2.75	0.95	1.30
3	-0.2	29.1	3.17	4.31	3.00	4.37	5.54	8.00	5.77	8.06
4	-16.4	-25.4	5.23	7.21	6.22	8.87	2.17	3.27	4.37	6.09
...	...									

Regression example: earthquake epicenter

- Preprocessing: normalize all measurements using the largest time recorder
- Network inputs: $\vec{t}^T = \begin{pmatrix} t_p^1 & t_s^1 & t_p^2 & t_s^2 & t_p^3 & t_s^3 & t_p^4 & t_s^4 \end{pmatrix}$
- Network architecture: $k = 3$ hidden layers with $m = 64$ nodes in each layer

$$\vec{z}_0 = \sigma(\mathbf{W}_0\,\vec{t} + \vec{b}_0) \qquad \vec{z}_1 = \sigma(\mathbf{W}_1\,\vec{t} + \vec{b}_1) \qquad \vec{z}_2 = \sigma(\mathbf{W}_2\,\vec{t} + \vec{b}_2)$$

- Network output: two nodes (latitude, longitude)

$$\vec{z} = \begin{pmatrix} x \\ y \end{pmatrix} = \mathbf{W}_3\,\vec{z}_2(\vec{z}_1(\vec{z}_0(\vec{t})))$$

  - Where $\mathbf{W_3}$ is the $2 \times m$ matrix containing the weights from the last hidden layer to the two output nodes

Regression example: earthquake epicenter

- For optimizing the adjustable parameters $\mathbf{W}$ and $\vec{b}$, we use the mean squared error (MSE) as our objective function

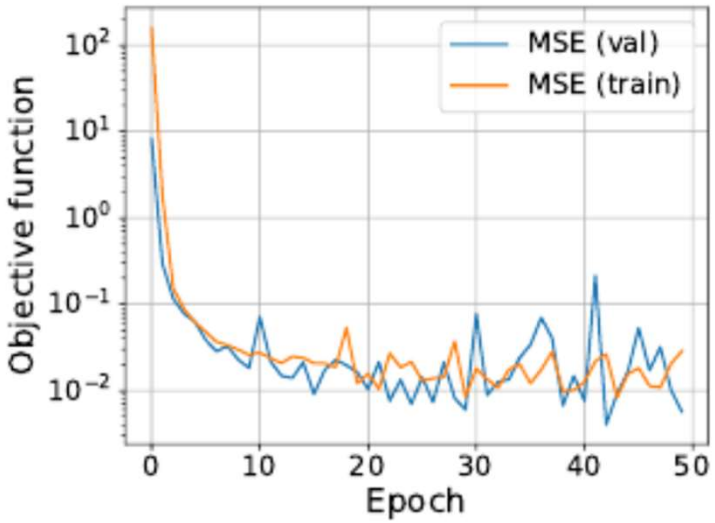
$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2]$$

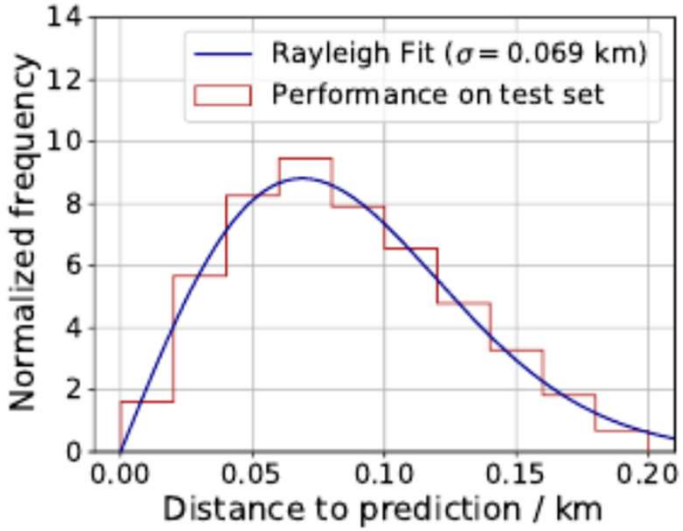
- MSE is calculated using only a minibatch
  - we assume a minibatch of size $n = 32$

- Initialization of $\mathbf{W}$ is done by sampling from a normal distribution with standard deviation: $\sigma = \sqrt{2/(n_{in} + n_{out})}$ (Glorot / Xavier normal)

- While we set $\vec{b}$ to zero

Regression example: earthquake epicenter





- Network training with 19k simulated earthquakes
- Validation with 1k data
- Performance improves with number of epochs
- Model testing: mean distance between true and predicted locations is about 70m!

Classification example: image classification

- Input: black-and-white 32×32 images of spiral galaxies arranged into a vector $\vec{x}_i$ of size $n = 32 \times 32 = 1024$
- Target variable: rotational orientation of the spirals
- Vector of each image is fed to the 1024 nodes of the input layer
- Architecture: similar to previous example (regression) – main differences:
  - Different size of input layer
  - Predict classes instead of continuous variables

Classification example: image classification

- Two possibilities for output layer:

  1. Single node: scalar output turned into probability using sigmoid activation function ($z > 0.5$ and $z < 0.5$)

  2. Two nodes connected in such a way that resulting values z_1 and z_2 are normalized to $z_1 + z_2 = 1$

     - We go for the second option in this example!

Classification example: image classification

- **Softmax** function: $\vec{z} = \begin{pmatrix} \hat{z}_1 \\ \hat{z}_2 \end{pmatrix} = \frac{1}{e^{z_1}+e^{z_2}} \begin{pmatrix} e^{z_1} \\ e^{z_2} \end{pmatrix}$

- Target variable (galaxy orientation): $\vec{y}_j^T = (y_1 \quad y_2) = \begin{cases} (1 \quad 0) \ \ Right \\ (0 \quad 1) \ \ Left \end{cases}$
  - Known as **one-hot-encoding**

- Objective function (cross-entropy): $\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m} (y_{i,1} \quad y_{i,2}) \begin{pmatrix} \log(\vec{z}_1(\vec{x}_i)) \\ \log(\vec{z}_2(\vec{x}_i)) \end{pmatrix}$

Challenges of training

- Example: Parameter update in multi-layer network
  - Consider:
    - ANN with three layers and one node per layer
    - Linear-mapping activation function: $\sigma(x) = x,\ b = 0$
  - At iteration m, we have:
    - Prediction at iteration m: $z^{[m]} = x\, W_1^{[m]}\, W_2^{[m]}\, W_3^{[m]}$
    - Objective function (MSE): $\mathcal{L} = [\hat{z}(x) - z^m(x)]^2$
    - Gradient of W_1: $g_1 = \frac{\partial \mathcal{L}}{\partial W_1} = 2[\hat{z}(x) - x\, W_1^{[m]}\, W_2^{[m]}\, W_3^{[m]}]\, W_2^{[m]}\, W_3^{[m]}$

      - Gradients depend on all other weights
      - Depending on numerical values, gradients could tend to zero or infinity:
        - **vanishing** or **exploding gradient** – prevents successful training

Challenges of training

- Example: Parameter update in multi-layer network
  - Consider:
    - ANN with three layers and one node per layer
    - Linear-mapping activation function: $\sigma(x) = x,\ b = 0$
  - At iteration $m + 1$: parameters are updated according to respective gradients g_i and other parameters
    - Coordinating all parameter updates is usually not performed due to comp. inefficiency
    - Instead we assume the network's overall update can be factorized into independent updates of the individual parameters
    - Resulting in unplanned shifts in the network activations – **internal covariate shifts**

Proceedings of the 32nd International Conference on Machine Learning 37, pp. 448–456 (2015)

Challenges of training

- Example: Parameter update in multi-layer network
  - Factorization assumption leads to additional challenge: How to choose appropriate learning rate α
  - Problem can be evident by expanding $m+1$ iteration in terms of Taylor on α:

$$z^{m+1} = x\, W_1^{[m+1]}\, W_2^{[m+1]}\, W_3^{[m+1]} = x(W_1^{[m]} + \alpha\, g_1^{[m]})(W_2^{[m]} + \alpha\, g_2^{[m]})(W_3^{[m]} + \alpha\, g_3^{[m]})$$

$$z^{m+1} = \ \ldots - x\, \alpha\, g_1^{[m]} W_2^{[m]} W_3^{[m]} + \ \ldots$$

  - Choice of an appropriate learning rate α is challenging:
    - it depends on the parameters of the deeper layers
    - parameters and their gradients areall interconnected

Challenges of training

- Example: Parameter update in multi-layer network
    - Above example is oversimplification – in reality use nonlinear activation functions
    - Activation functions with saturation, such as sigmoid functions, can naturally lead to vanishing gradients
        - ➢ The fact that for sigmoid $\sigma(x) < 1$ could potentially result into $\sigma^n(x) \to 0$
- Ways to stabilize training process:
    1. Residuals training and improved gradient propagation
        - ➢ Network architecture modified using **shortcuts**
    2. Batch normalization
        - ➢ Minibatch data transformed before entering subsequent layer
    3. Self-normalizing neural networks
        - ➢ cause the activation averages and variances to converge to fixed points

Residuals learning

- There is a hierarchy in the learning process:
  - First layers of a network try to learn a rough structure of the desired mapping $f(x)$
  - The deeper layers learn small detailed modifications of the rough mapping
  - An extreme case: deeper layers learn the identity mapping

  - More convenient approach: instead of learning something very close to the identity mapping, learn deviations form it

$$f(\vec{x}) = \mathbf{1}\vec{x} + \delta(\vec{x})$$

    - Learn residuals: $\delta = f - \mathbf{1}$

  - Mechanism to implement the identity mapping: **shortcut**

    - input $\vec{x}$ is element-wise added to the mapping $\delta(\vec{x})$

Residuals learning

- For two layers we have:

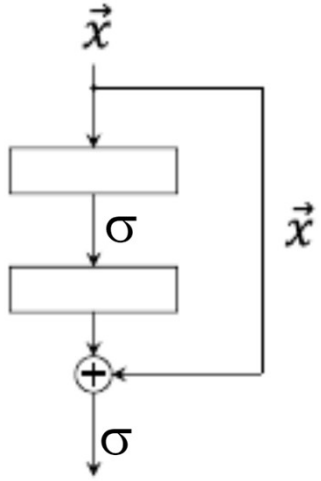
$$z = \sigma(\mathbf{W}_2 \left[\sigma\left(\mathbf{W}_1 \vec{x} + \vec{b}_1\right)\right] + \vec{b}_2 + \vec{x})$$

- Shortcut method adds only negligible computational cost
- Number of parameters remains unchanged

- If dimensions of $\vec{x}$ and $\delta(\vec{x})$ are different, an appropriate linear projector is used instead of the identity mapping:

$$f(\vec{x}) = \mathbf{W}_s \vec{x} + \delta(\vec{x})$$

- Shortcuts method has been successfully used in computer vision (ResNet, DenseNet)



K. He, X. Zhang, S. Ren and J. Sun, Deep residual learning for image recognition, (2015)

Batch normalization

- **Batch normalization** refers to a re-parametrization method applied between network layers
  - for a minibatch m of size k, the outputs y (for each node) are collected and transformed as follows:

$$\mu_m = \frac{1}{k}\sum_{i=1}^{k} y_i \qquad \sigma_m = \sqrt{\epsilon + \frac{1}{k}\sum_{i=!}^{k}(y_i - \mu_m)^2} \qquad y_i' = \frac{y_i - \mu_m}{\sigma_m}$$

  - Additional step: re-enable dispersion and shift:

$$y_j' = \gamma_j \frac{y_i - \mu_m}{\sigma_m} + \beta_j$$

  - Parameters γ_j and β_j help perform targeted changes of the averages and variances, not accessible through standard network design

S. Ioffe and C. Szegedy, Batch normalization: Accelerating deep net-work training by reducing internal covariate shift, Proceedings of the 32ndInternational Conference on Machine Learning 37, pp. 448–456 (2015)

Batch normalization

- Advantages of **batch normalization** have been empirically demonstrated

  1. It accelerates training of very deep networks

  2. Ensures moderate parameter values and gradients

  3. Using batch normalization, the choice of **learning rate** "α" becomes less critical

Self-normalization

- Stability of neural networks can be also achieved through engineered activation functions
- This allows for **self-normalizing networks**, defined through the requirement that mean and variance of the activation distribution of a layer can be mapped onto the corresponding mean and variance of the next layer

- Consider outputs of activations x_i of network layer j with n nodes:

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad v_j = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \mu_j\right)^2$$

- Mapping of moments of layer j to $j+1$: $\left(\frac{\mu_{j+1}}{v_{j+1}}\right) = K\left(\frac{\mu_j}{v_j}\right)$

Self-normalization

- Idea of mapping K:
  - First specify affine mapping at each node i of layer $j+1$: $y_i = \sum_k W_{jk}\, x_k + b_i$
  - For simplicity, assume input variables x_k and parameters W_{jk}, b_i to be independent random variables
  - For large number of nodes in layer $j+1$, due to the central limit theorem, the affine mapping y_i follows normal distribution
  - The affine mapping is followed by the activation function.

Self-normalization

- Idea of mapping K:
  - The moments of the activation distribution, for $j+1$ layer, are calculated via integration (convolutions):

$$\mu_{j+1} = \int_{-\infty}^{\infty} \sigma(y)\, p_G(y)\, dy$$

$$\nu_{j+1} = \int_{-\infty}^{\infty} \sigma^2(y)\, p_G(y)\, dy - \mu_{j+1}^2$$

  - Where p_G is normal (Gaussian) distribution
  - properties of the mapping K are determined by the properties of the activation function

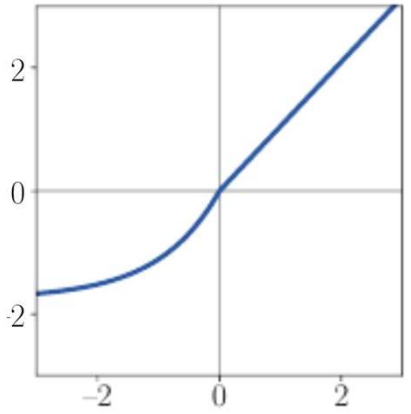
Self-normalization

- How to design $\sigma(y)$:
  - Aspects of $\sigma(y)$ to keep in mind:

    1. Positive and negative values are needed for adjusting μ_{j+1}
    2. Gradients greater than 1 needed to increase ν_{j+1}
    3. Saturation needed to decrease ν_{j+1}
    4. Continuous curve needed for converging to fixed points of moments

  - Suitable choice: **SELU** (scaled exponential linear unit)

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



  - similar to the exponential linear unit (ELU), discussed previously
  - which provides gradients with improved normalization capacities
  - robust saturation regime to prevent a randomized on- and off-switching of activations

G. Klambauer et al., Self-normalizing neural networks, arXiv e-prints (2017)

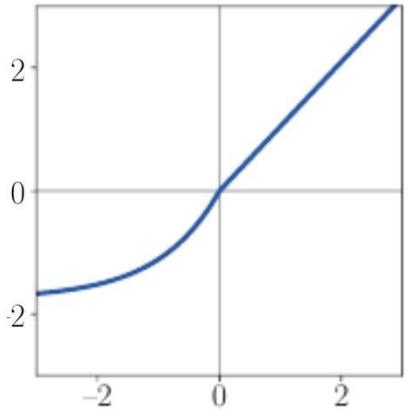
Self-normalization

- How to design $\sigma(y)$:
  - Aspects of $\sigma(y)$ to keep in mind:

    1. Positive and negative values are needed for adjusting μ_{j+1}
    2. Gradients greater than 1 needed to increase ν_{j+1}
    3. Saturation needed to decrease ν_{j+1}
    4. Continuous curve needed for converging to fixed points of moments

  - Suitable choice: **SELU** (scaled exponential linear unit)

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



  - SELU adds also two parameters λ and α
  - These parameters can be adjusted to fulfill requirements such as convergence toward fixed points
  - For initialization based on standard normal distribution, typical values: $\alpha = 1.6733,\ \lambda = 1.0507$

G. Klambauer et al., Self-normalizing neural networks, arXiv e-prints (2017)

Summary

- Fully-connected networks correspond to a standard network architecture suitable for different tasks
- For prediction of continuous values MSE is one choice of objective function
- For prediction of multidimensional continuous values, the last layer holds n output nodes
- For predicting categories (*classes*), the number of output values corresponds to the number of classes - output values are further processed using the *softmax* function that guarantees their interpretation as probabilities
- The objective function of choice for classification is *cross-entropy*

Summary

- Training of *deep* neural networks is challenging
  - Training needs to be robust against *vanishing/exploding* gradients
  - Challenges in selecting appropriate *learning rates*
  - Iterative updates of many parameters simultaneously, in an uncoordinated way, leads to problems known as *internal covariate shifts*

- Methods of network stabilization:
  - One way is to optimize networks by learning *residual* mappings by modifying the network architecture using *shortcuts*
  - Another way is to perform *batch normalization,* i.e. to re-normalize the outputs of a layer before it enters the next one
  - A third option is to use *self-normalizing networks,* where the distributions of activations are iteratively stabilized. This is achieved using a suitable choice of objective function (e.g. *SELU)*