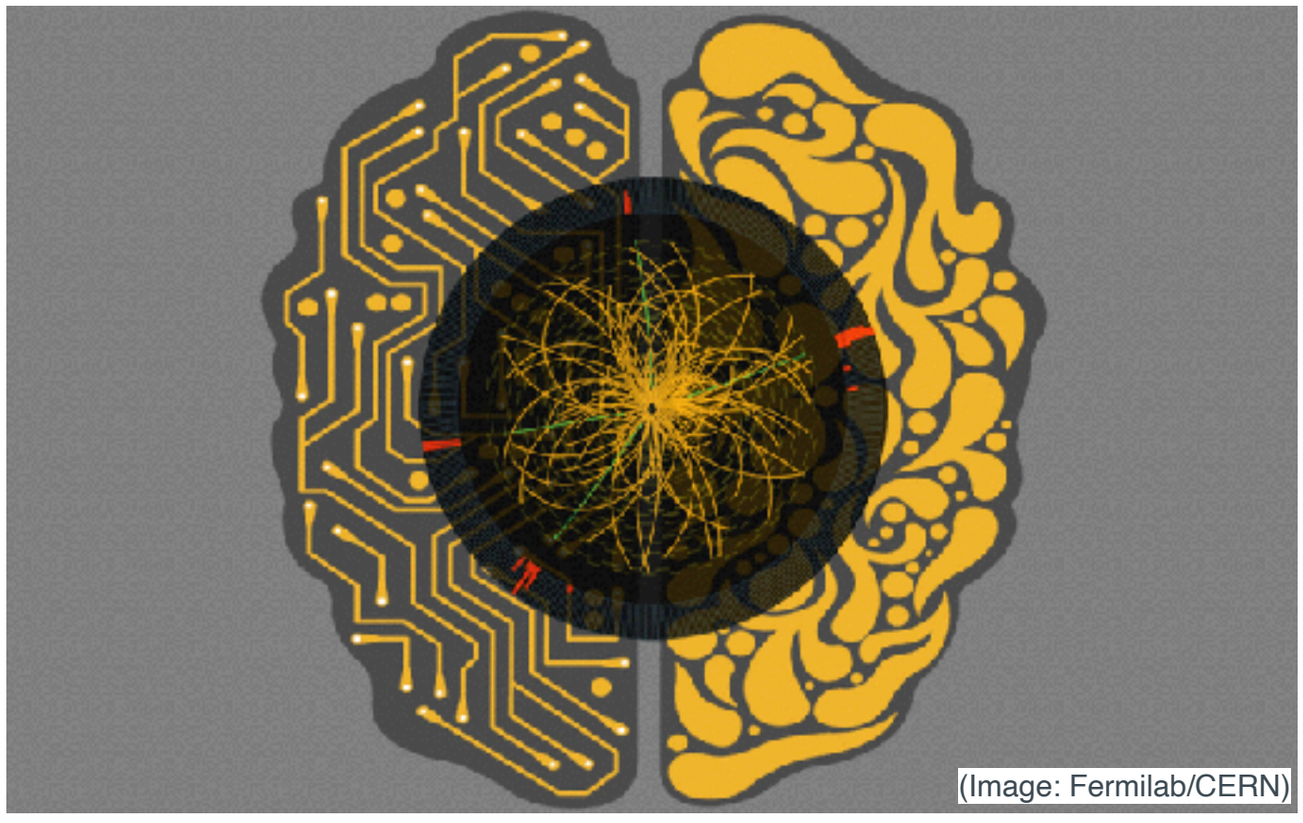
PHY 835: Collider Physics Phenomenology

Machine Learning in Fundamental Physics

Gary Shiu, UW-Madison



Lecture 10: Feedforward Neural Networks

Recap of Lecture 9

- Kernel methods
- Soft margins
- Ensemble Methods
  - Bagging
  - Boosting
  - Random Forest

Outline for today

- Feedforward neural networks
- Backpropagation
- Regularization

References: Deep Learning Book, 1803.08823

Neural Networks: History

- Old subject (e.g., Hebbian learning, Rosenblatt 1958 Perceptron, …)
- Revived interest since ~2010 due to:
- Performance increase on classification tasks via trained deep neural networks
- Calculations on GPUs possible
- Open source packages and implementations: Tensorflow, Pytorch, Keras, ….

NEURIPS participants (started in 1987)

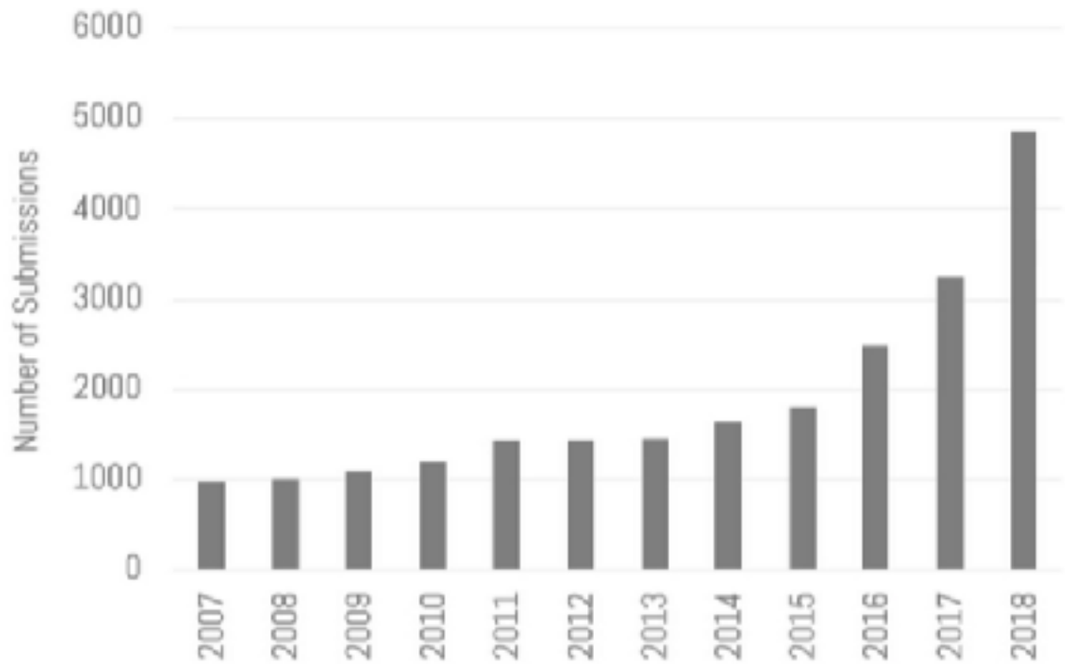


Image: https://medium.com/syncedreview/neurips-2018-through-the-eyes-of-first-timers-5156384900bd

Neural Networks (NNs)

- Neural Networks can be used for different ML tasks, e.g.,
  - General purpose NNs for supervised learning
  - NNs designed specifically for image processing, the most prominent e.g. being Convolutional Neural Networks (CNNs)
  - NNs for sequential data e.g. Recurrent Neural Networks (RNNs)
  - NNs for unsupervised learning e.g. Deep Boltzmann Machines.

- High-level libraries & packages, e.g., **Caffe: https://caffe.berkeleyvision.org**

  Keras: https://keras.io

  Pytorch: https://pytorch.org

  TensorFlow: https://www.tensorflow.org

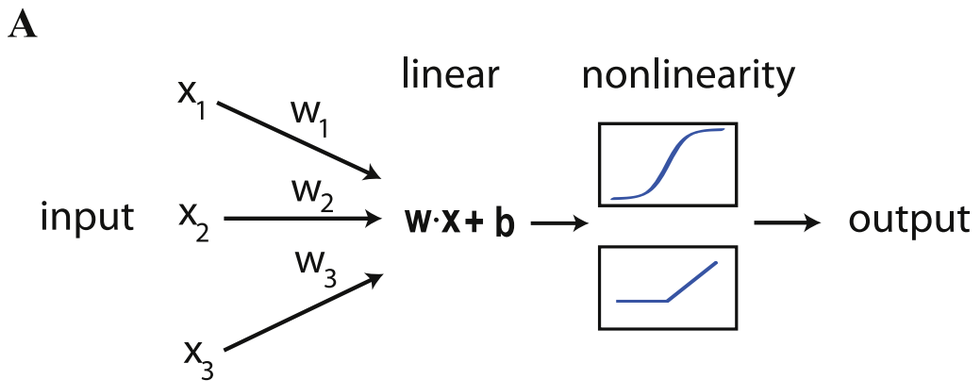
Neural Network Basics

- Neural networks are neural-inspired nonlinear models for ML (powerful extensions of linear & logistic regression, softmax, …)

- Basic unit is a neuron:

Linear operation $z^{(i)} = \boldsymbol{w}^{(i)} \cdot \boldsymbol{x} + b^{(i)} = \mathbf{x}^T \cdot \mathbf{w}^{(i)},$

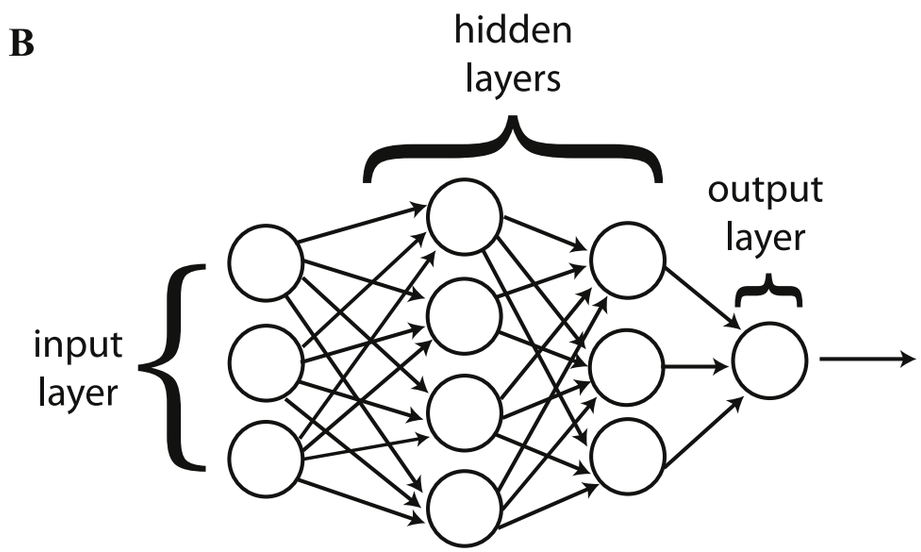
Non-linearity $a_i(\mathbf{x}) = \sigma_i(z^{(i)}),$



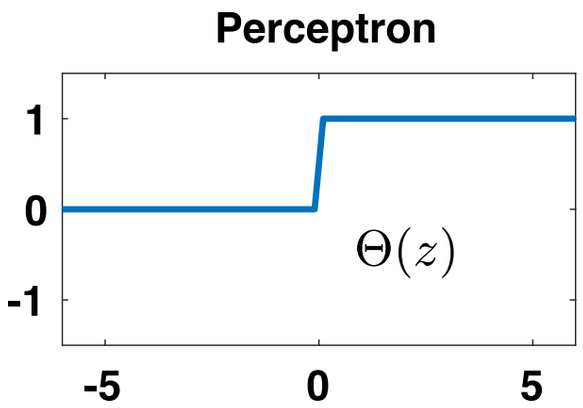
- A NN consists of many neurons stacked into layers:

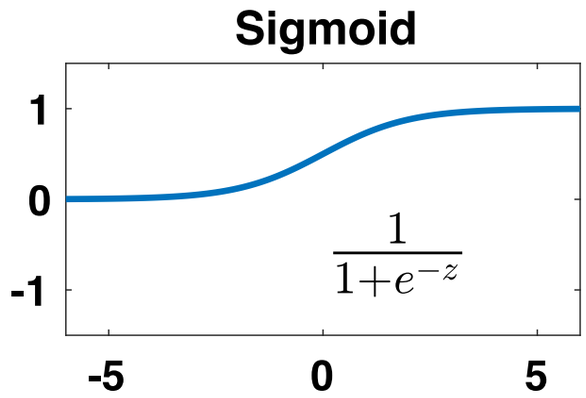
Outputs of the input layer
= inputs of the next *hidden* layer

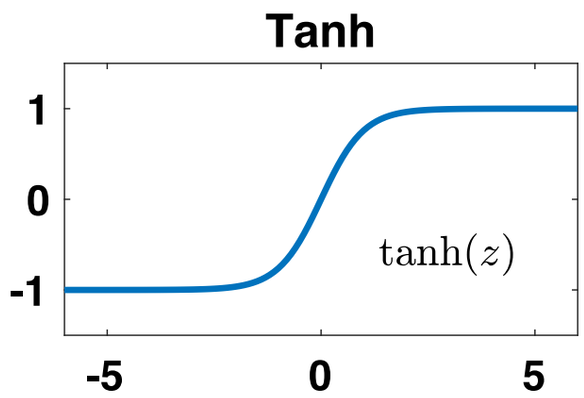
The output layer is often a simple classifier
e.g., soft-max function or a linear regression

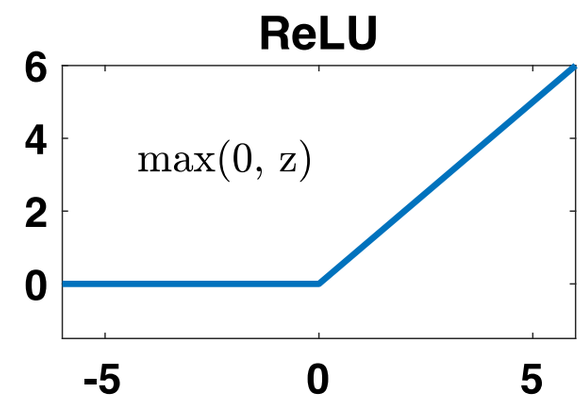


Activation Function

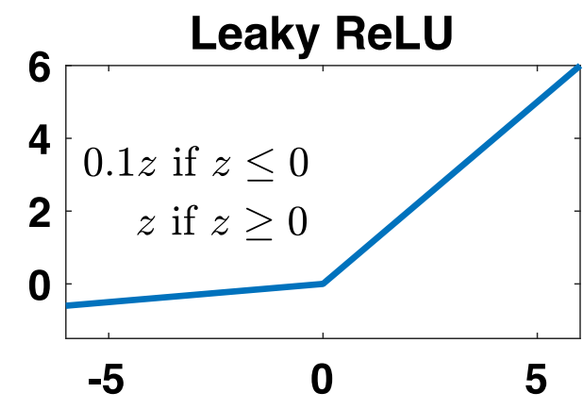


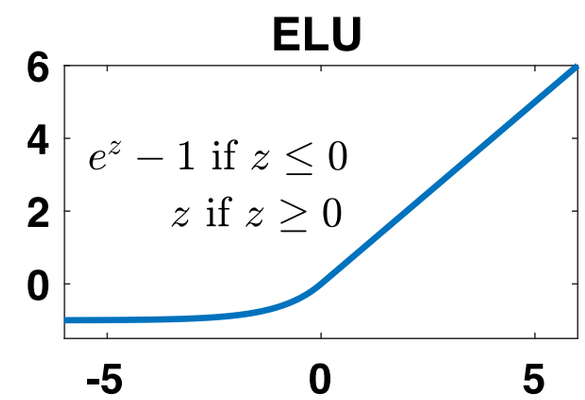






Rectified Linear Unit





Exponential Linear Unit

Train neural nets using gradient descent.

Perceptron: derivative is either zero or infinite (not suitable)
Sigmoid, Tanh: differentiable but has the drawback of saturation

$$\partial\sigma/\partial z \to 0 \text{ for } z \gg 1.$$

ReLU, Leaky ReLU, ELU: can overcome this vanishing gradient problem.

Network Architecture

- The way in which neurons are layered and connected is known as the network architecture.

- **Hidden layers:** often multiple layers (deep networks)

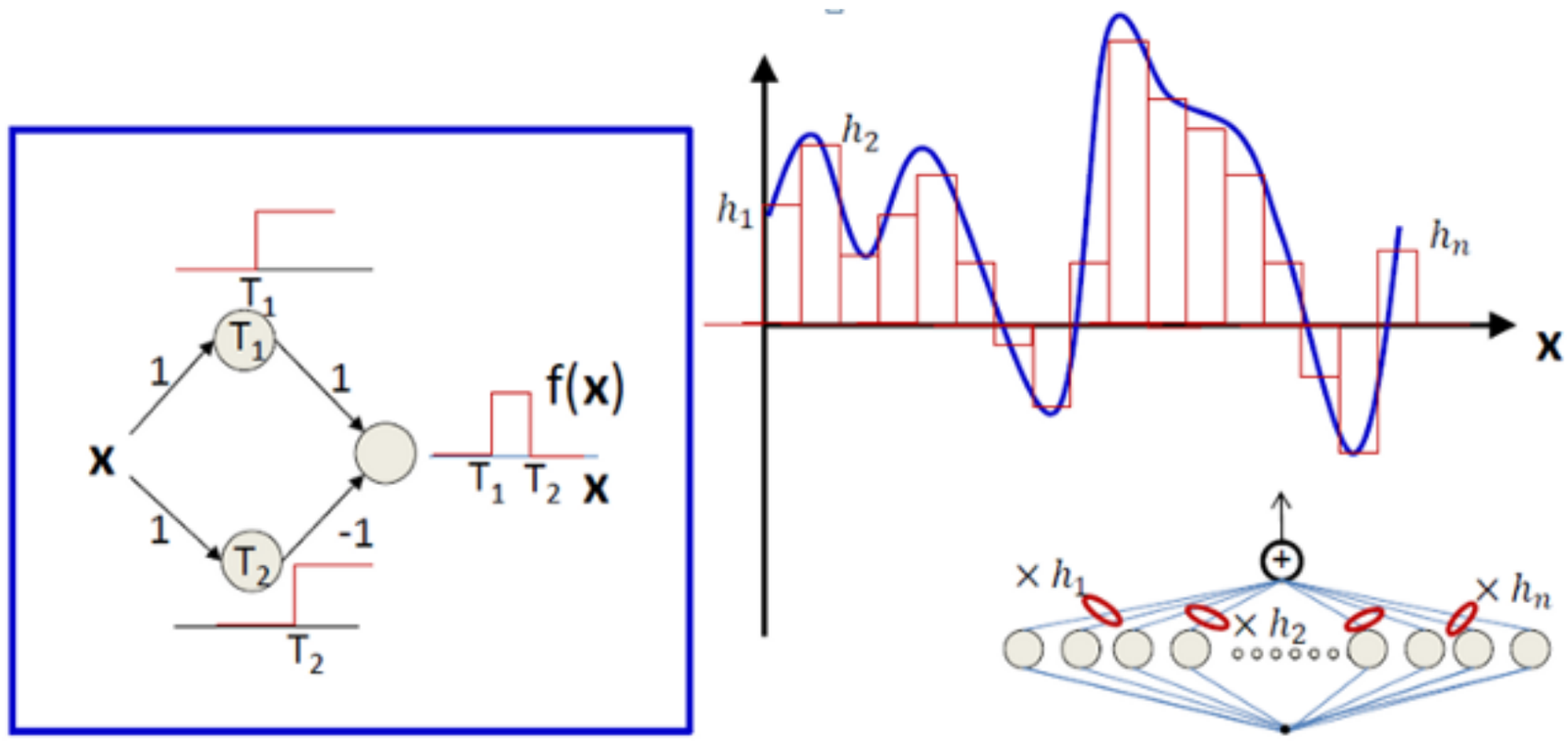
- **Size of layer (i.e. how many nodes needed):** depends on problem (hyperparameter)

- Design depends on the task, the amount and type of data. Certain architectures are easier to train, others are better in learning complicated features.

- Greatly expands the representational power (expressivity) compared with a simple softmax or linear regression.

Universal Approximation Theorem (1989)

Neural network with a single hidden layer can approximate any continuous, multi-input/multi-output functions with arbitrary accuracy.



A visual proof that neural nets can compute any function
http://neuralnetworksanddeeplearning.com/chap4.html

Training Deep Networks

- Construct a loss function:

Mean sq. error (linear regression)

$$E(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i(\mathbf{w}))^2,$$

all the weights and biases in the network

Cross-entropy (binary classification)

$$E(\mathbf{w}) = -\sum_{i=1}^{n} y_i \log \hat{y}_i(\mathbf{w}) + (1 - y_i)\log\left[1 - \hat{y}_i(\mathbf{w})\right].$$

Softmax (multi-class classification)

$$E(\mathbf{w}) = -\sum_{i=1}^{n}\sum_{m=0}^{M-1} y_{im}\log \hat{y}_{im}(\mathbf{w}) + (1 - y_{im})\log\left[1 - \hat{y}_{im}(\mathbf{w})\right].$$

$$y_{im} = \begin{cases} 1, & \text{if } y_i = m \\ 0, & \text{otherwise.} \end{cases}$$

- Use gradient descent to minimize the cost function and find the optimal weights and biases.
- Neural networks contain multiple hidden layers that make taking the gradient computationally more difficult (backpropagation).

Backpropagation

- Need to calculate as many gradients as parameters at each step of the gradient descent.
- The backpropagation algorithm exploits the layered structure of neural networks to more efficiently compute gradients.
- Systematic use of chain rule (summarized by 4 equations).

- Assume there are L layers in the network, $l = 1, \ldots, L$.
- Denote by w^l_{jk} the weight for the connection from the k-th neuron in layer $l-1$ to the j-th neuron in layer l (whose bias is b^l_j).
- For feedforward network, the activation a^l_j of the j-th neuron:

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right) = \sigma(z^l_j), \qquad \textbf{linear weighted sum:} \quad z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j.$$

Backpropagation

- The cost function E depends *directly* on the activities of the output layer a_j^L, and *indirectly* on the activities of neurons in lower layers.
- Define the error of the j-th neuron in the l-th layer:

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial a_j^l}\sigma'(z_j^l), \qquad \text{(I)}$$

- This error can alternatively be interpreted as:

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial E}{\partial b_j^l}, \qquad \text{(II)}$$

- Using chain rule:

$$\begin{aligned}\Delta_j^l = \frac{\partial E}{\partial z_j^l} &= \sum_k \frac{\partial E}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \Delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \left(\sum_k \Delta_k^{l+1} w_{kj}^{l+1}\right)\sigma'(z_j^l). \qquad \text{(III)}\end{aligned}$$

Backpropagation

- The last backpropagation comes from differentiating the cost function w.r.t. the weight:

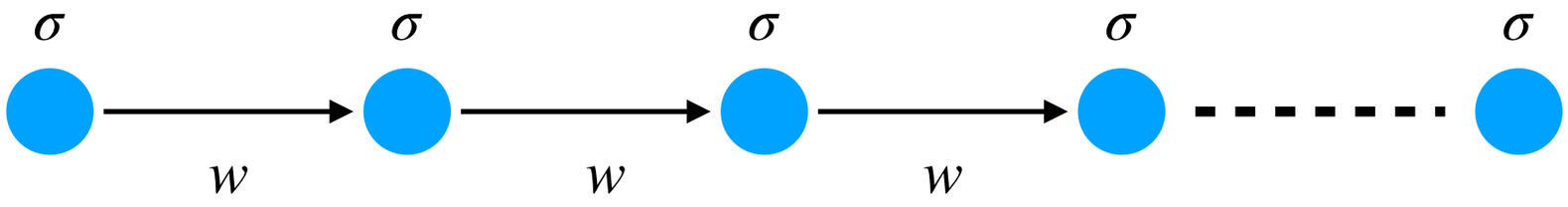
$$\frac{\partial E}{\partial w_{jk}^l} = \frac{\partial E}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1} \qquad \text{(IV)}$$

The Backpropagation Algorithm

1. **Activation at input layer:** calculate the activations a_j^1 of all the neurons in the input layer.
2. **Feedforward:** starting with the first layer, exploit the feed-forward architecture through Eq. (123) to compute z^l and a^l for each subsequent layer.
3. **Error at top layer:** calculate the error of the top layer using Eq. (I). This requires to know the expression for the derivative of both the cost function $E(\mathbf{w}) = E(\boldsymbol{a}^L)$ and the activation function $\sigma(z)$.
4. **"Backpropagate" the error:** use Eq. (III) to propagate the error backwards and calculate Δ_j^l for all layers.
5. **Calculate gradient:** use Eqs. (II) and (IV) to calculate $\frac{\partial E}{\partial b_j^l}$ and $\frac{\partial E}{\partial w_{jk}^l}$.

Problem of Vanishing or Exploding Gradients

- Illustrate this problem with a network with 1 neuron per layer



- Backpropagating the error (assuming equal weights):

$$\Delta_j^1 = \Delta_j^L \prod_{j=0}^{L-1} w\sigma'(z_j) = \Delta_j^L (w)^L \prod_{j=0}^{L-1} \sigma'(z_j) \quad = \Delta_j^L (w\sigma_0')^L$$

- The errors and their gradients vanish/blow-up unless $w\sigma_0' \approx 1$.

- Avoid vanishing and explosion of gradients by using appropriate initialization of weights and regularization (e.g. gradient clipping, batch normalization), and using non-linearities that do not saturate.

- Finding ways to solve this problem is an active area of research.

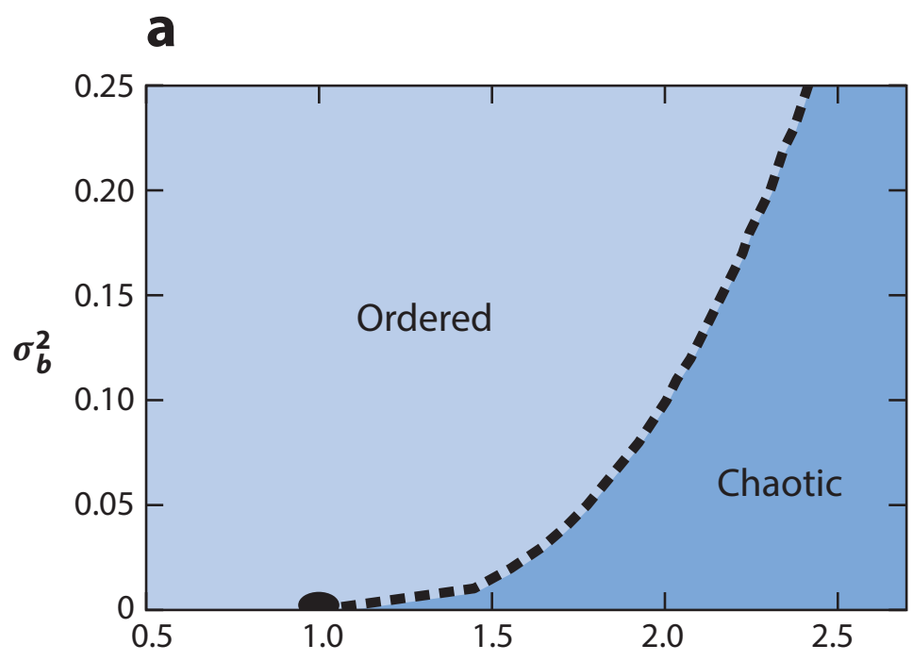
Phases of Weights

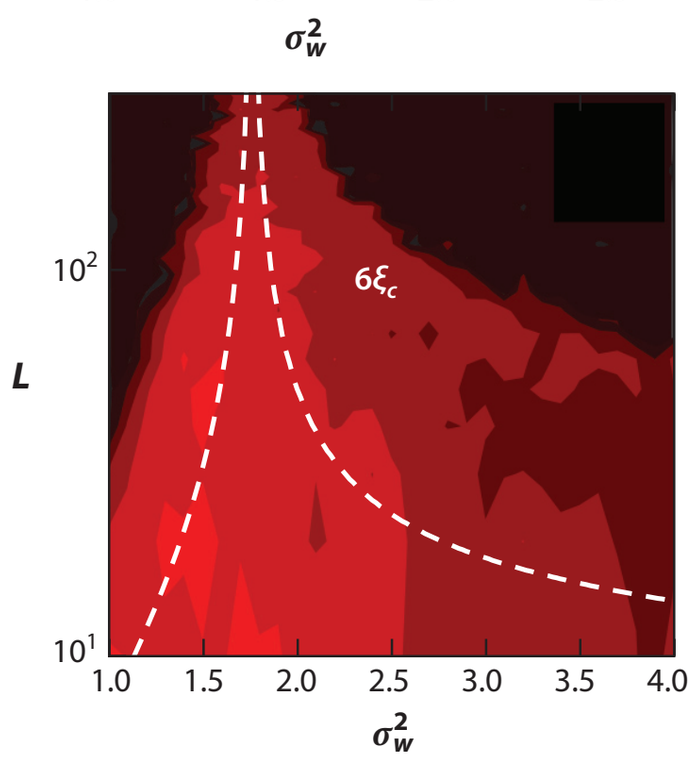
- Trainable vs non-trainable depends on weight initialization.

- Chaotic regime: network not trainable (explosion of gradients), system is unstable.

- Ordered regime: potentially vanishing gradients.

- Training near phase boundary most efficient.



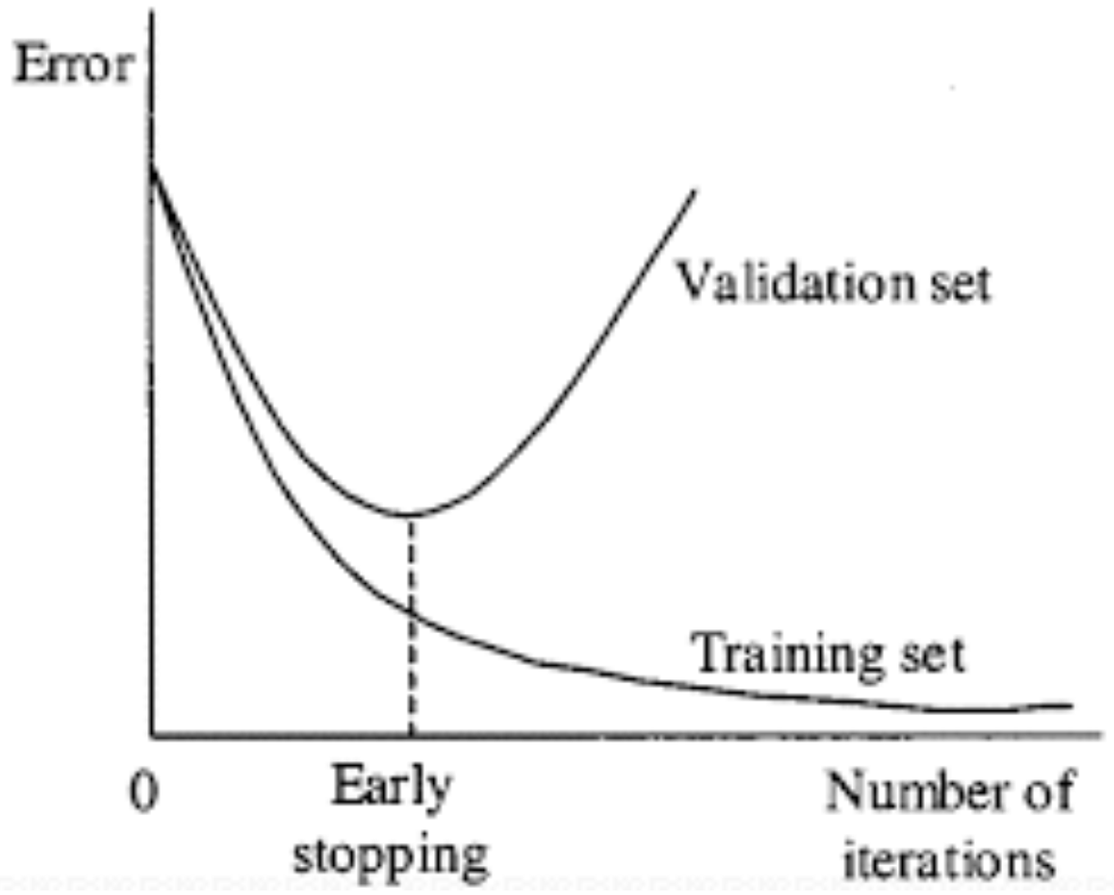


Bahri et al, Statistical Mechanics of Deep Learning: https://www.annualreviews.org/doi/pdf/10.1146/annurev-conmatphys-031119-050745

Regularizing Neural Networks

Implicit Regularization of SGD

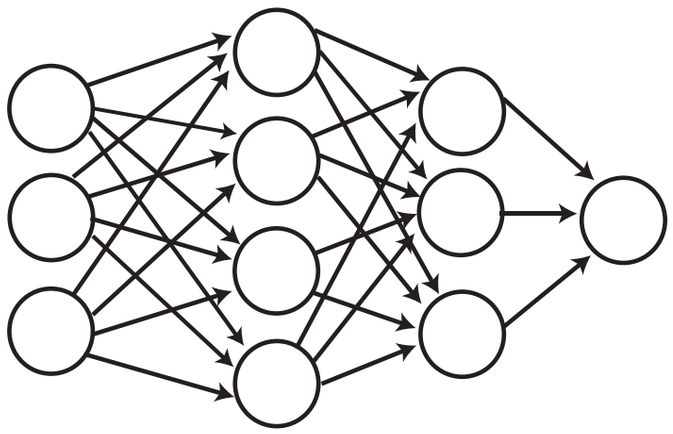
- **Initialization:** random weights (to break leftover symmetries), Gaussianly distributed (experiment with different variances).
- **Hyperparameter tuning:** search for optimal values on a log scale.
- **Early stopping:** divide training data into a training set (larger) and a validation set (smaller)



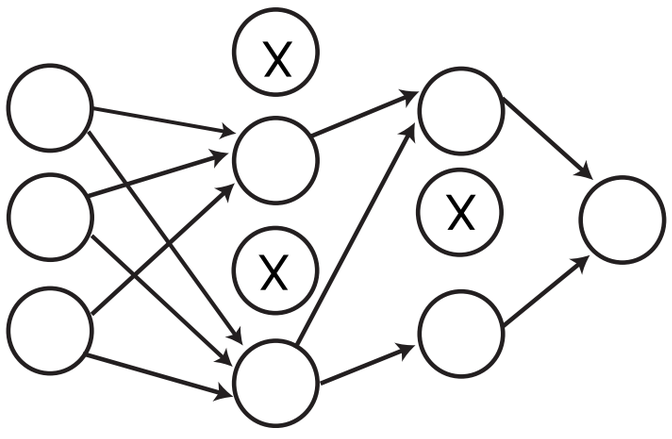
Dropout

- Prevent overfitting by reducing spurious correlations among neurons in network. Similar idea as in bagging in ensemble methods, but without using different datasets.
- Randomly dropping out neurons (and their connections) during each step of training. Gradient descent only performed on this subset of neurons. Predictions made with all neurons active.

Standard Neural Net



After applying Dropout



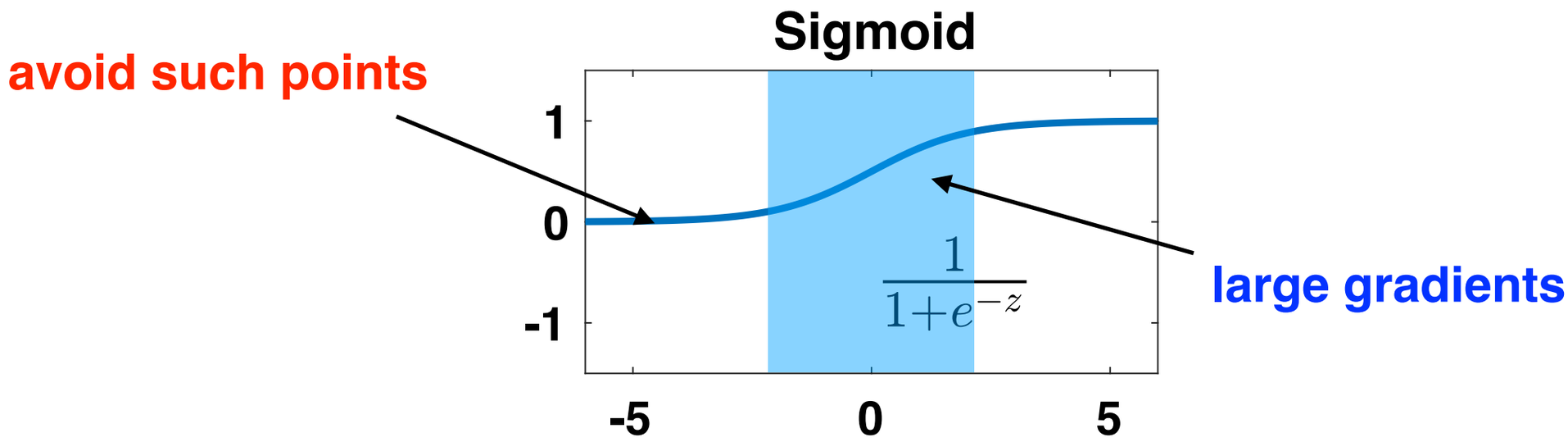
weights are present only a fraction p of the time, rescale weights:

$$\mathbf{w}_{\text{test}} = p\mathbf{w}_{\text{train}}.$$

Batch Normalization

Ioffe, Szegedy 1502.03167 (~17000 citations)
c.f. the Higgs paper (~11000 citations); AdS/CFT (~15000 citations)

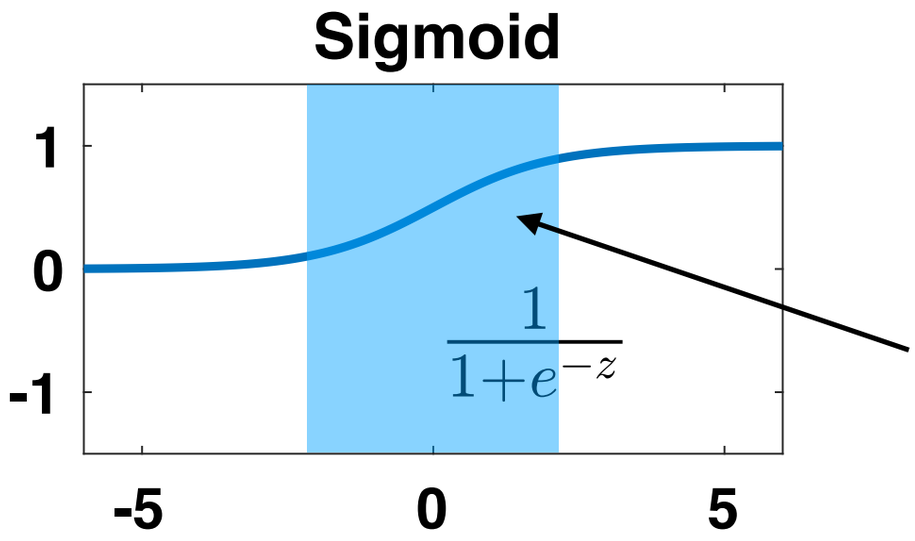
- Observation: training works best when inputs are centered around zero with respect to bias (wx+b) for activation like tanh, sigmoid:



- Batch Normalization: additional layers which standardize inputs by the mean and variance of the mini-batch.

Batch Normalization

- **Standardize inputs:**



$$z_k^l \longrightarrow \hat{z}_k^l = \frac{z_k^l - \mathbb{E}[z_k^l]}{\sqrt{\mathrm{Var}[z_k^l]}},$$

average and variance over all samples in mini-batch in layer l

Keep this info. A drawback is that it forces the network to live purely in the linear region around $z = 0$

- **Scale and shift inputs with learnable parameters:**

$$\hat{z}_k^l \longrightarrow \hat{\mathrm{z}}_k^l = \gamma_k^l \hat{z}_k^l + \beta_k^l.$$

- Initialize the NN with standard inputs at the beginning of training. Backpropagation then adjusts γ and β during training.
- Advantage: improves learning speed, acts as a regularizer.

Deep Learning Packages

- **Kersas:** High-level framework, less control over the operations in between the layers.

http://physics.bu.edu/~pankajm/ML-Notebooks/HTML/NB11_CIX-DNN_mnist_Keras.html

- TensorFlow (supported by Google): construct data flow graphs with nodes (activations) and edges (data array).

http://physics.bu.edu/~pankajm/ML-Notebooks/HTML/NB12_CIX-DNN_ising_TFlow.html

- Pytorch: controls over the inter- and intra-layer operations w/o the need to introduce computational graphs

http://physics.bu.edu/~pankajm/ML-Notebooks/HTML/NB13_CIX-DNN_susy_Pytorch.html

SUSY Dataset

Experiment with Pytorch in Notebook 13

http://physics.bu.edu/~pankajm/ML-Notebooks/HTML/NB13_CIX-DNN_susy_Pytorch.html

Adjust hyperparameters e.g. learning rate and training set size.

Optimal performance at the edge of the grid, extend the grid size to obtain better results.

1. **Collect and pre-process the data.**
2. **Define the model and its architecture.**
3. **Choose the cost function and the optimizer.**
4. **Train the model.**
5. **Evaluate and *study* the model performance on the test data.**
6. **Use the validation data to adjust the hyperparameters (and, if necessary, network architecture) to optimize performance for the specific dataset.**

Summary

- Feedforward neural networks
- Backpropagation
- Implementation of neural networks with Keras
- Regularization