DEEP NEURAL NETWORKS

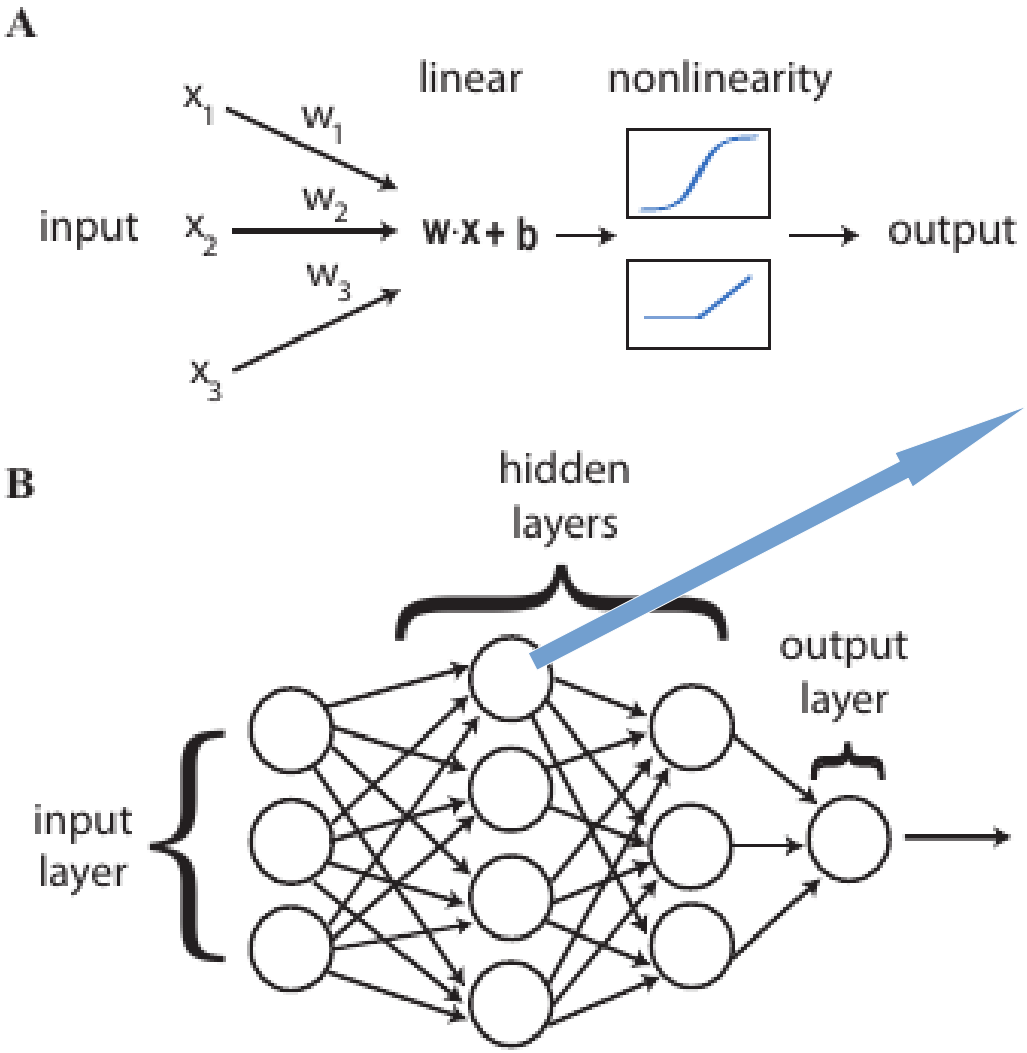
Material

- NB11_CIX-DNN_mnist_Keras.ipynb
  NB12_CIX-DNN_ising_TFlow.ipynb
  NB13_CIX-DNN_susy_Pytorch.ipynb
- We are going to write our own example

DNN Python packages

- **Keras**: simpler, less flexible
- **TensorFlow**: steeper learning curve
- **Pytorch**: GPU, fast array operations

- **Caffe**: in C++



$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

Nonlinear activation of neuron *j* in layer *l*

Loss function E(w's, b's)

FIG. 34 **Basic architecture of neural networks.** (A) The basic components of a neural network are stylized neurons consisting of a linear transformation that weights the importance of various inputs, followed by a non-linear activation function. (b) Neurons are arranged into layers with the output of one layer serving as the input to the next layer.

Nonlinear activation of neurons

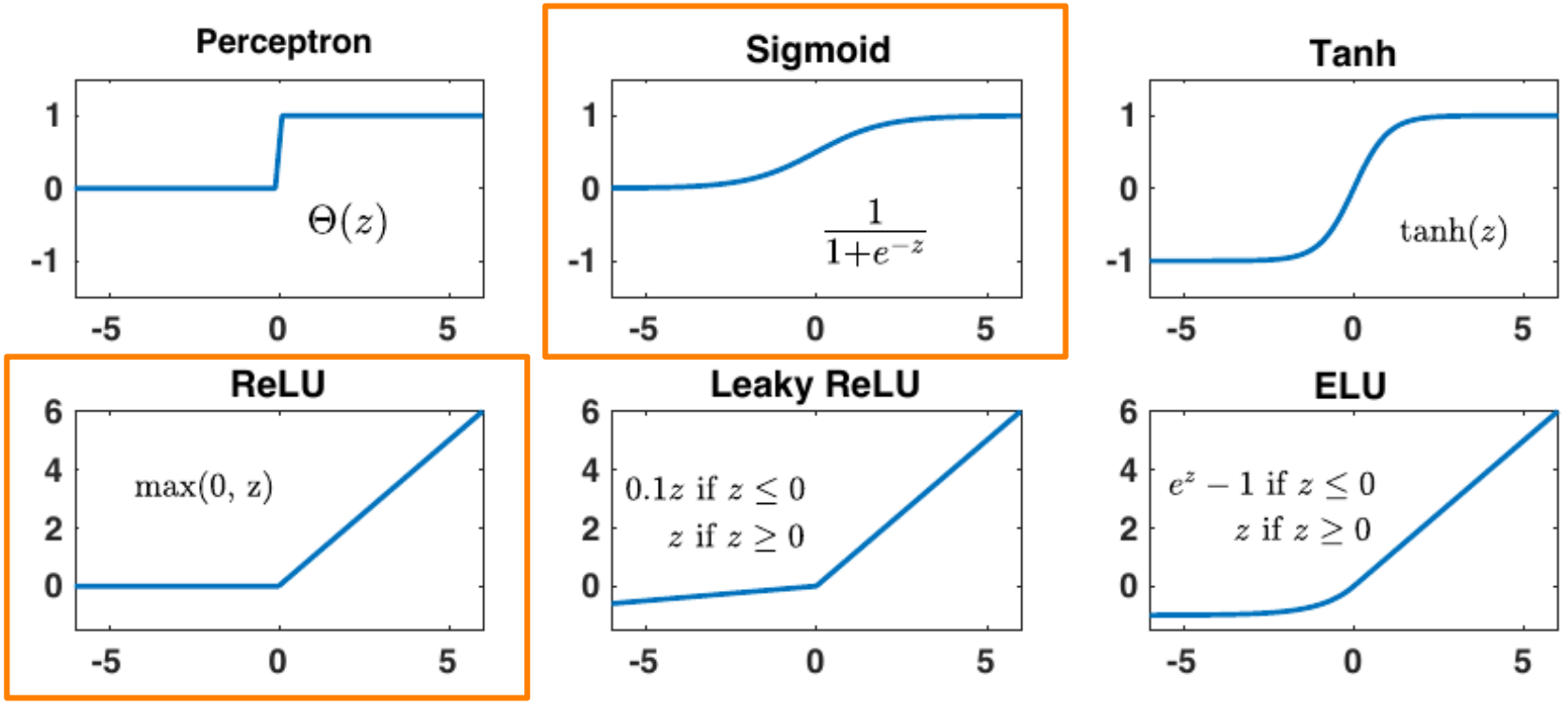


FIG. 35 **Possible non-linear activation functions for neurons.** In modern DNNs, it has become common to use non-linear functions that do not saturate for large inputs (bottom row) rather than saturating functions (top row).

Supervised learning

- DNN give answers, data with labels:

  discriminative

  … as opposed to generative models

  (unsupervised learning in the last two lectures)

- Training is still:

  cost/loss function + gradient descent minimization

  (more complex achitecture than linear regression, etc.)

  via backpropagation

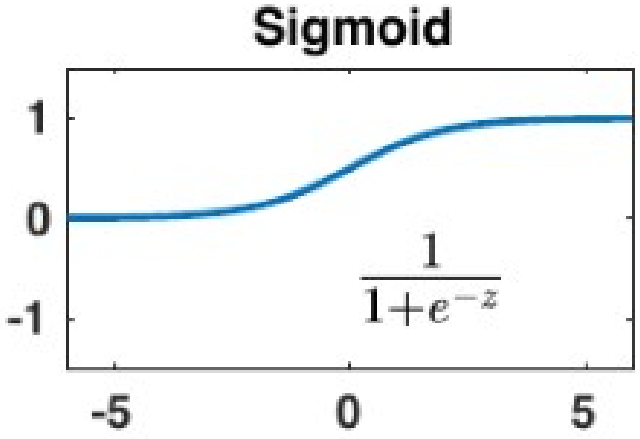
DNN

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 hyper-parameters / architecture to optimize performance for the specific dataset.

From the review by Mehta et al.

1) Collect and pre-process the **data**

- Pre-process:
  - Remove averages
  - Rescale to unit variance
  - Rearrange data
- Divide:



DNN

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 hyper-parameters / architecture to optimize performance for the specific dataset.

From the review by Mehta et al.

2) Define the **model** and its **architecture**

- Avoid vanishing/exploding gradients
  - Use suitable activation functions
  - Correctly initialize the network parameters

(a factor of 2 can lead to exploding gradients)

On a more technical and mathematical note, another problem that occurs in deep networks, which transmit information through many layers, is that gradients can vanish or explode. This is, appropriately, known as *the problem of vanishing or exploding gradients*. This problem is especially pronounced in neural networks that try to capture long-range dependencies, such as Recurrent Neural Networks for sequential data. We can illustrate this problem by considering a simple network with one neuron in each layer. We further assume that all weights are equal, and denote them by w. The behavior of the backpropagation equations for such a network can be inferred from repeatedly using Eq. (III):

$$\Delta_j^1 = \Delta_j^L \prod_{j=0}^{L-1} w\sigma'(z_j) = \Delta_j^L (w)^L \prod_{j=0}^{L-1} \sigma'(z_j), \qquad (126)$$

where Δ_j^L is the error in the L-th topmost layer, and $(w)^L$ is the weight to the power L. Let us now also assume that the magnitude $\sigma'(z_j)$ is fairly constant and we can approximate $\sigma'(z_j) \approx \sigma_0'$. In this case, notice that for large L, the error Δ_j^1 has very different behavior depending on the value of $w\sigma_0'$. If $w\sigma_0' > 1$, the errors and the gradient blow up. On the other hand, if $w\sigma_0' < 1$ the errors and gradients vanish. Only when the weights satisfy $w\sigma_0' \approx 1$ and the neurons are not saturated will the gradient stay well behaved for deep networks.

This basic behavior holds true even in more complicated networks. Rather than considering a single weight, we can ask about the eigenvalues (or singular values) of the weight matrices w_{jk}^l. In order for the gradients to

DNN

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 hyper-parameters / architecture to optimize performance for the specific dataset.

From the review by Mehta et al.

3) Choose **cost function** and **optimizer**

- Continuous cost / discrete classification
- Gradient descent method
- Try different learning rates (η). Recipes:
  - Rates over five logarithmic grid points
  - Optimal is ~1/10 of rate that would lead to explosion
- Quick pre-training with small dataset in real time

Batch Normalization is a regularization scheme that has been quickly adopted by the neural network community since its introduction in 2015 (Ioffe and Szegedy, 2015). The basic inspiration behind Batch Normalization is the long-known observation that training in neural networks works best when the inputs are centered around zero with respect to the bias. The reason for this is that it prevents neurons from saturating and gradients from vanishing in deep nets. In the absence of such centering, changes in parameters in lower layers can give rise to saturation effects in higher layers, and vanishing gradients. The idea of Batch Normalization is to introduce additional new "BatchNorm" layers that standardize the inputs by the mean and variance of the mini-batch.

Consider a layer l with d neurons whose inputs are $(z_1^l, \ldots, z_d^l)$. We standardize each dimension so that

$$\hat{z}_k^l = \frac{z_k^l - E[z_k^l]}{\sqrt{\text{Var}[z_k^l]}}, \tag{127}$$

where the mean and expectation are taken over all samples in the mini-batch. One problem with this procedure is that it may change the representational power of the neural network. For example, for tanh non-linearities, it may force the network to live purely in the linear regime around $z = 0$. Since non-linearities are crucial to the representational power of DNNs, this could dramatically alter the power of the DNN. For this reason, one introduces two new parameters γ_k^l and β_k^l for each neuron that can shift and scale the normalized input

$$\hat{y}_k^l = \gamma_k^l \hat{z}_k^l + \beta_k^l. \tag{128}$$

These new parameters are then learned just like the weights and biases using backpropagation (since this is just an extra layer for the chain rule). We initialize the neural network so that at the beginning of training the inputs are being standardized. Backpropagation than adjusts γ and β during training.

In practice, Batch Normalization considerably improves the learning speed by preventing gradients from vanishing. However, it also seems to serve as a powerful regularizer for reasons that are not fully understood.

DNN

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 hyper-parameters / architecture to optimize performance for the specific dataset.

From the review by Mehta et al.

4) Train the model

- Minibatches reduce overfitting
- Precision to achieve?
- Resources available (time, power, etc..)
- “Early stopping”
  - Training dataset → Training + validation datasets
  - When validation error starts to rise (due to overfitting), stop training

DNN

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 hyper-parameters / architecture to optimize performance for the specific dataset.

From the review by Mehta et al.

5) Evaluate and study the model performance on the **test** data.

- Accuracy
- Performance
- Overfitting

  large number of parameters compared to the data available

  → the network went too far with its attempt of describing specific features of that particular dataset

6)...**adjust**... to optimize performance...

- Tune the parameters
- Revisit choices of
  - Optimizer
  - Nonlinear units
  - Architecture
  - (even) data pre-processing
  - …

Whereas it is always possible to view Steps 1-5 as generic and independent of the particular problem we are trying to solve, it is only when these steps are put together in Step 6 that the real benefit of deep learning is revealed, compared to less sophisticated methods such as regression or bagging, see Secs. VI, VII, and VIII. The

Advantages / disadvantages of DNN

- See the review for more details...

Today: simple classification, DNN with Keras

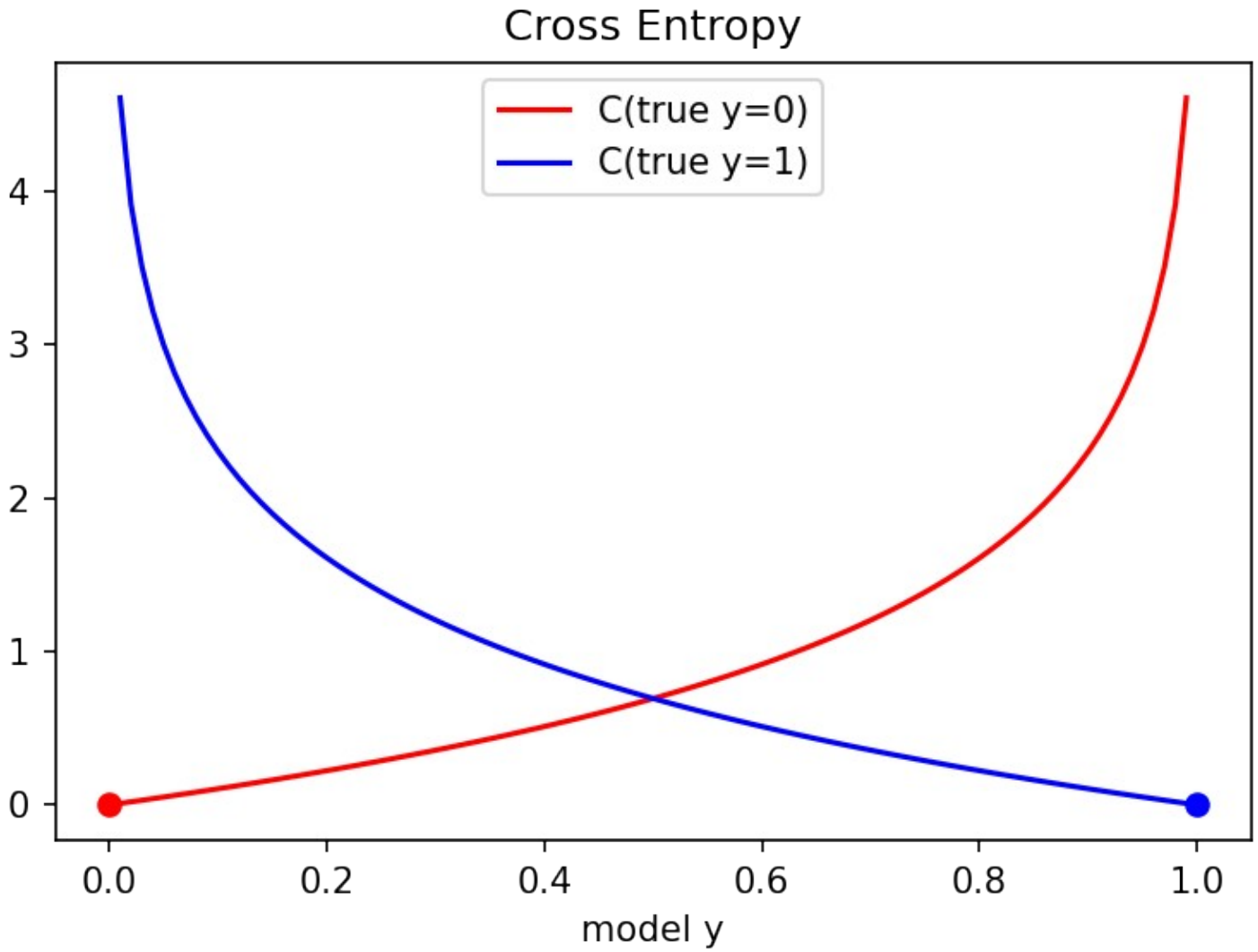
Binary Cross Entropy, C

(see e.g. Sect. 7)

- y=0,1
- Minibatch B
- $\hat{y}_i$ predicted by the model

  0,1 via the sigmoid function $\sigma(z)=\frac{1}{1+e^{-z}}$

$$C=\sum_{i\in B}\left[-y_i \ln \hat{y}_i -(1-y_i)\ln(1-\hat{y}_i)\right]$$


Cross Entropy
C(true y=0)
C(true y=1)
4
3
2
1
0
0.0
0.2
0.4
0.6
0.8
1.0
model y