Neural Networks

STAT3009 Recommender Systems

by Ben Dai (CUHK)
on October 30, 2024

» Recall LFM: NCF

Recall the basic Latent Factor Model:

$$\min_{\boldsymbol{P},\boldsymbol{Q}} \frac{1}{|\Omega|} \sum_{(u,i)\in\Omega} (r_{ui} - \mu - a_u - b_i - \boldsymbol{p}_u^\intercal \boldsymbol{q}_i)^2 + \lambda \left(\sum_{u=1}^{n} \|\boldsymbol{p}_u\|_2^2 + \sum_{i=1}^{m} \|\boldsymbol{q}_i\|_2^2 \right) \tag{1}$$

- The **interaction** between users and items is formulated as an inner product.
- It can be extended to model high-order nonlinear interactions.

» Nonlinear interaction: Neural networks

- For a general nonlinear function f, the predicted rating can be formulated as $\widehat{r}_{ui} = f(\boldsymbol{p}_u, \boldsymbol{q}_i)$.
- Examples of nonlinear methods include **polynomials**, **B-splines**, and **kernel methods**.
- Alternatively, $f(\cdot, \cdot)$ can be a **neural network**.

Before applying **neural networks** into recommender systems, we shall have a quick overview of machine learning models and neural networks.

» Recall ML overview

Data A pair of input features and its corresponding outcome, denoted as (feat, label).

→Model f_θ: a parameterized function that maps features to labels.

Loss $L(\cdot, \cdot)$: a measure of the difference between the predicted outcome and the true outcome.

→Opt The algorithm used to solve the problem.

→: data and loss remain the same; we design our model as a neural network and find an opt algorithm to solve it.

» Recall ML Overview

→Step 1 Design your **model**, including parameters and hyperparameters

→Step 2 Train parameters based on the training set with different hyperparameters

Step 3 Compute validation loss for each hyperparameter using a validation set or k-fold cross-validation; and select the optimal hyperparameters

Step 4 Refit the model with the optimal hyperparameters based on all data

Step 5 Make **predictions** for the test set

» Recall ML Overview

→Step 1 Design your **model**, including parameters and hyperparameters

→Step 2 Train parameters based on the training set with different hyperparameters

Step 3 Compute validation loss for each hyperparameter using a validation set or k-fold cross-validation; and select the optimal hyperparameters

Step 4 Refit the model with the optimal hyperparameters based on all data

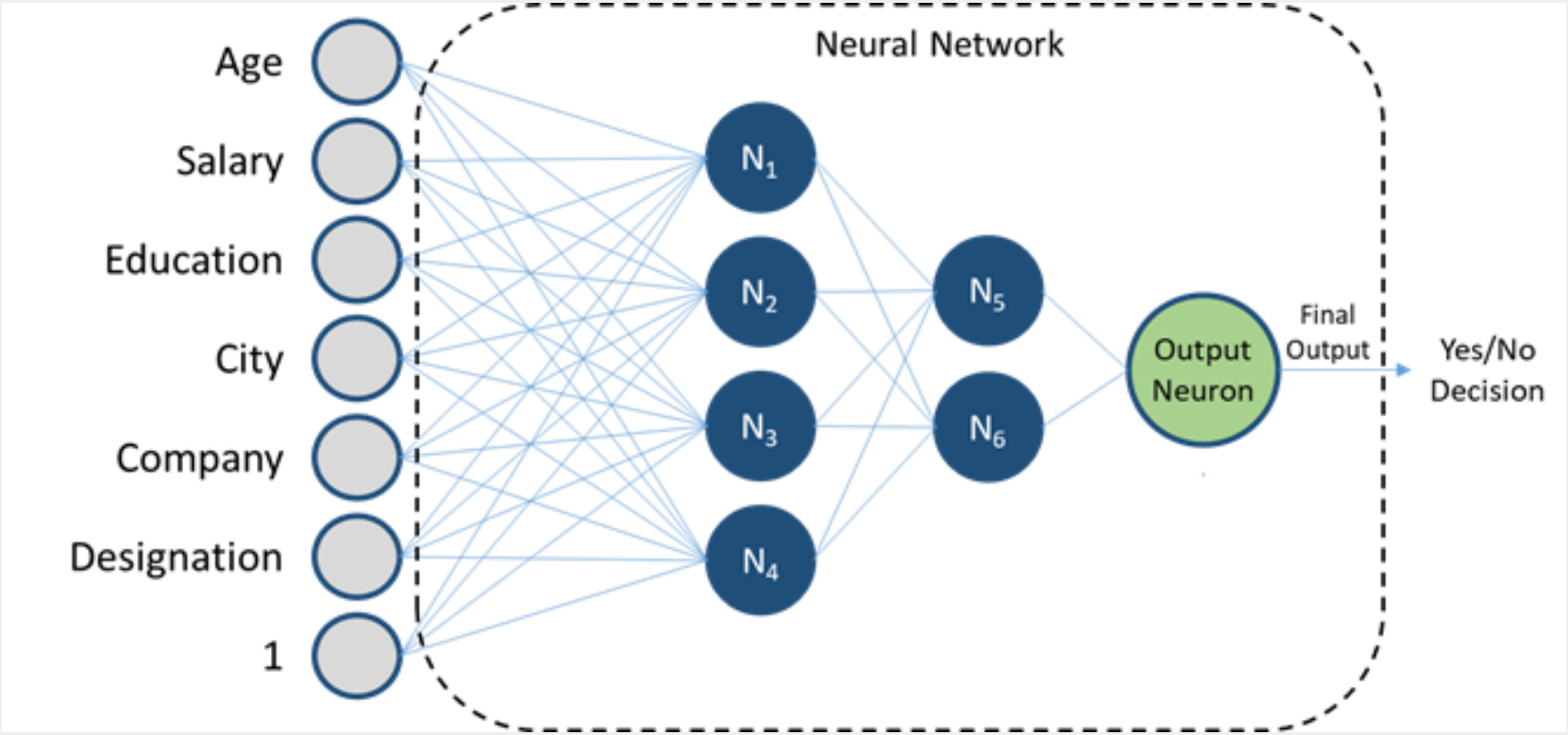
Step 5 Make **predictions** for the test set

Q1 What are the parameters and hyperparameters for a neural network?

Q2 How do we train a neural network?

Model *architecture*:

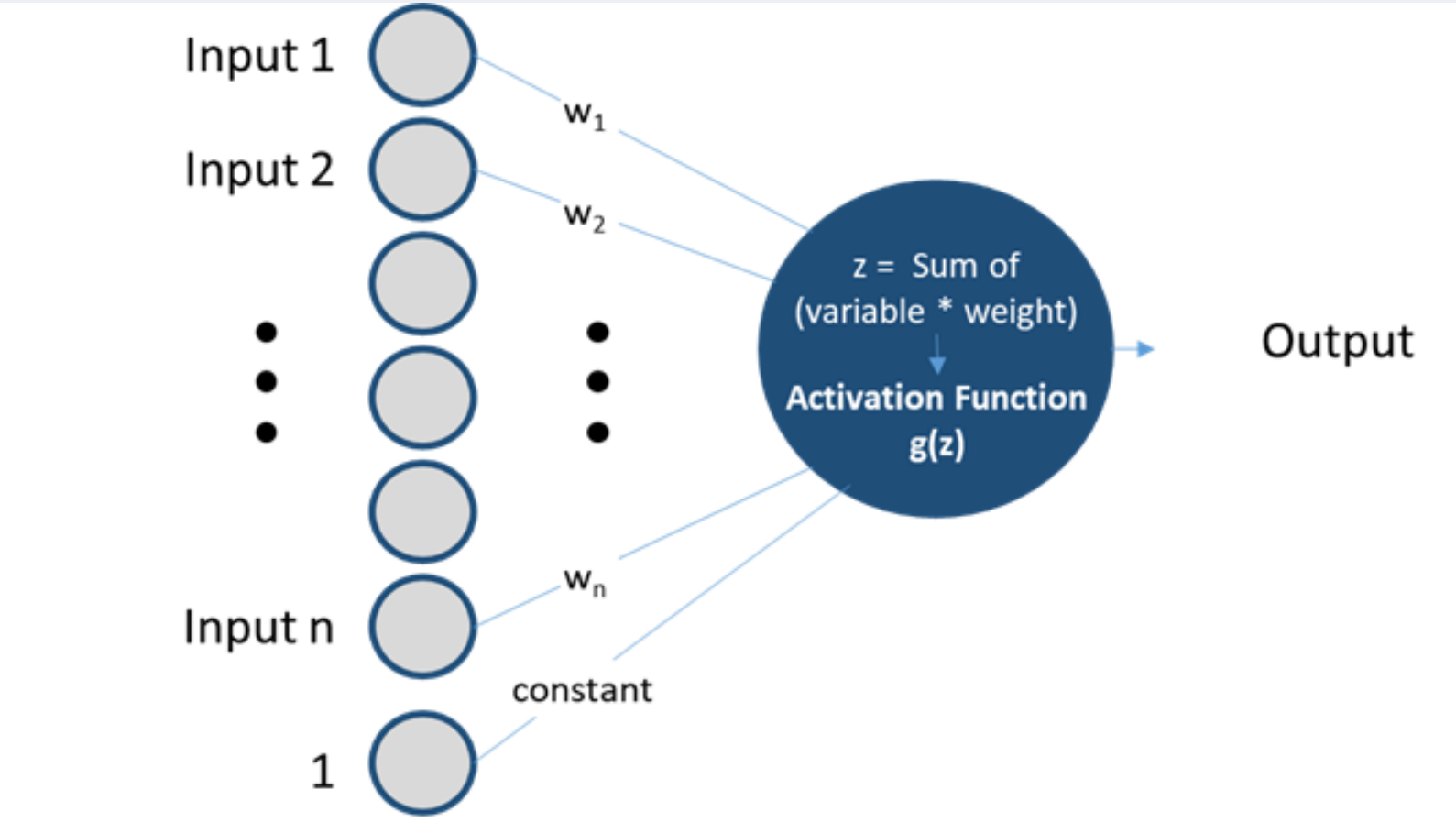
Input $\rightarrow$ Hidden Layer 1 $\rightarrow \cdots \rightarrow$ Hidden Layer L $\rightarrow$ Output



1

[1] https://towardsdatascience.com/deep-learning-101-neural-networks-explained-9fee25e8ccd3

Neuron diagram: Examining a single neuron in a subsequent layer



2

[2] https://towardsdatascience.com/deep-learning-101-neural-networks-explained-9fee25e8ccd3

⚠ Mathematical formulation:

- **Nonlinear** activation function combined with a **linear combination** of outputs from the previous layer
- From input $\mathbf{f}_0 = \mathbf{x}$ to output $\mathbf{f}_L(\mathbf{x})$:

$$\boldsymbol{f}_l(\boldsymbol{x}) = A(\boldsymbol{W}_l \boldsymbol{f}_{l-1}(\boldsymbol{x}) + \boldsymbol{b}_l), \quad l = 1, \cdots, L.$$

- $\mathbf{W}_l \in \mathbb{R}^{d_l \times d_{l-1}}$ – weight matrix for the l-th layer
- $\mathbf{b}_l \in \mathbb{R}^{d_l}$ – bias terms in the l-th layer
- L – number of layers or depth of the neural network
- $A(\cdot)$ – activation function
  - Examples of activation functions: logistic (sigmoid), ReLU, tanh, and others[3];
- $\mathbf{f}_l(\mathbf{x}) \in \mathbb{R}^{d_l}$ – number of neurons in the l-th layer

[3] https://en.wikipedia.org/wiki/Activation_function

A1. Distinguishing between parameters and hyperparameters

Params The collection of all weights and biases,

$$\theta = \{\boldsymbol{W}_0, \boldsymbol{b}_0, \cdots, \boldsymbol{W}_{L-1}, \boldsymbol{b}_{L-1}\}$$

- **Weight matrices**: $\boldsymbol{W}_l \in \mathbb{R}^{d_l \times d_{l-1}}$, **bias vectors**: $\boldsymbol{b}_l \in \mathbb{R}^{d_l}$

A1. Distinguishing between parameters and hyperparameters

Params The collection of all weights and biases,

$$\theta = \{\boldsymbol{W}_0, \boldsymbol{b}_0, \cdots, \boldsymbol{W}_{L-1}, \boldsymbol{b}_{L-1}\}$$

- **Weight matrices**: $\boldsymbol{W}_l \in \mathbb{R}^{d_l \times d_{l-1}}$, **bias vectors**: $\boldsymbol{b}_l \in \mathbb{R}^{d_l}$

hp The architectural design of a neural network

- L - number of layers or depth of the neural network
- d_l - number of neurons in the l-th layer; $l = 1, \cdots, L$

Tradeoff As L and d_l increase,
the model becomes more complex
model complexity increases
training error decreases

A2. Training a neural network using Stochastic Gradient Descent (SGD) and backpropagation

SGD Recall. Compute stochastic gradients for all model parameters

* Gradient:

$$\frac{\partial \text{Loss}}{\partial \theta} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial L\big(y_i, \mathbf{f}_L(\mathbf{x}_i)\big)}{\partial \theta}$$

* Approximation using one sample:

$$\frac{\partial \text{Loss}}{\partial \theta} \leftarrow \frac{\partial L\big(y_i, \mathbf{f}_L(\mathbf{x}_i)\big)}{\partial \theta}$$

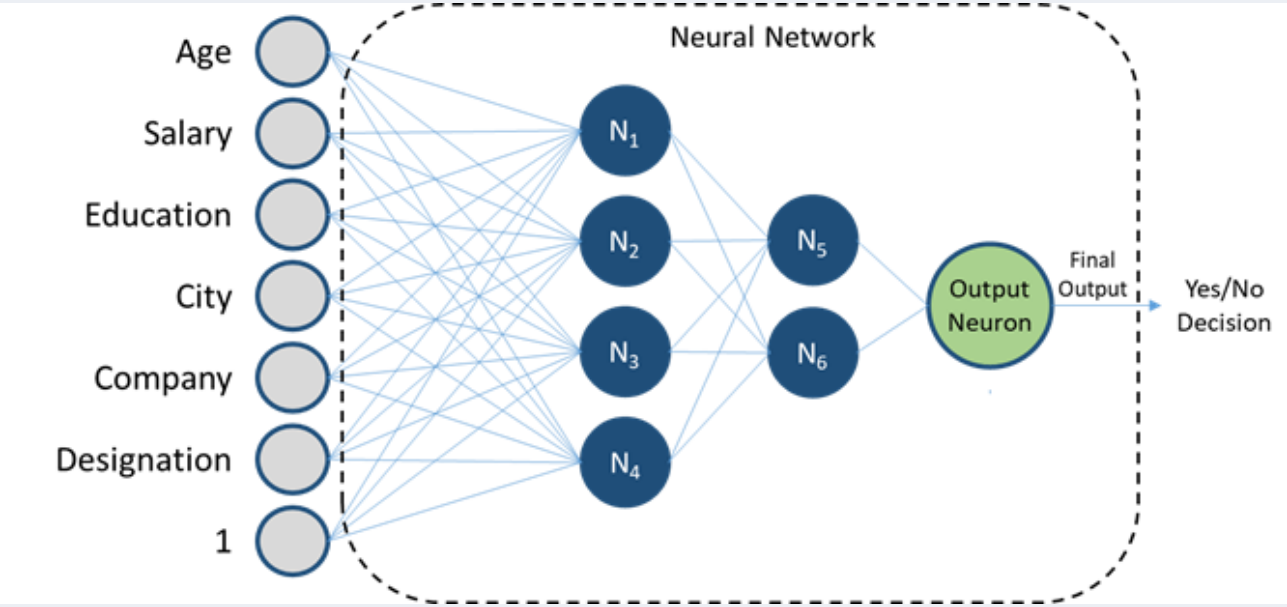
* Approximation using a mini-batch of samples

$$\frac{\partial \text{Loss}}{\partial \theta} \leftarrow \frac{1}{|Batch|} \sum_{i \in Batch} \frac{\partial L\big(y_i, \mathbf{f}_L(\mathbf{x}_i)\big)}{\partial \theta}$$

A2. Training a neural network using Stochastic Gradient Descent (SGD) and backpropagation

SGD Computing the stochastic gradient of $L(y_i, \mathbf{f}_L(\mathbf{x}_i))$ with respect to all model parameters

- Proceeding from the output layer (easiest) to the input layer (hardest)
- This process is known as backpropagation
- Reference: How the backpropagation algorithm works



» Backpropagation: Chain Rule (Optional)

Computing gradients for model parameters in different layers

Last layer $$\frac{\partial L(y_i, \mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{W}_L} = \frac{\partial L(y_i, \mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{f}_L(\mathbf{x}_i)} \frac{\partial \mathbf{f}_L(\mathbf{x}_i)}{\partial \mathbf{W}_L}$$

Layer L-1 $$\frac{\partial L(y_i, \mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{W}_{L-1}} = \frac{\partial L(y_i, \mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{f}_L(\mathbf{x}_i)} \frac{\partial \mathbf{f}_L(\mathbf{x}_i)}{\partial \mathbf{f}_{L-1}(\mathbf{x}_i)} \frac{\partial \mathbf{f}_{L-1}(\mathbf{x}_i)}{\partial \mathbf{W}_{L-1}}$$

Layer L-2 $$\frac{\partial L(y_i, \mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{W}_{L-2}} = \frac{\partial L(y_i, \mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{f}_L(\mathbf{x}_i)} \frac{\partial \mathbf{f}_L(\mathbf{x}_i)}{\partial \mathbf{f}_{L-1}(\mathbf{x}_i)} \frac{\partial \mathbf{f}_{L-1}(\mathbf{x}_i)}{\partial \mathbf{f}_{L-2}(\mathbf{x}_i)} \frac{\partial \mathbf{f}_{L-2}(\mathbf{x}_i)}{\partial \mathbf{W}_{L-2}}$$

$\dots$

Application of the chain rule!

Stochastic Gradient Descent (SGD) involves additional hyperparameters

$$\theta^{\text{new}} \leftarrow \theta^{\text{old}} - \text{learning rate} \times \sum_{i \in Batch} \frac{\partial L(y_i, \mathbf{f}_L(\mathbf{x}_i))}{\partial \theta}\Big|_{\theta^{\text{old}}}$$

- Learning rate - the step size for each gradient update
- Batch size - the number of samples used for each gradient update
- Number of epochs - the number of times the model is trained on the entire training dataset

- **Advantages:** flexible computing platforms, such as TensorFlow + Keras, are available for implementing custom neural networks.
- What we will do in practice?
  - Model definition. Specify your **custom model** $f(x)$
  - Loss and metrics. Define the **loss function** and **evaluation metrics** for the problem.
  - Optimization. Utilize tf.keras.optimizer.SGD, which will **automatically compute the gradient** via backpropagation[4]
  - Feed the **training data** to the defined model.

[4] http://neuralnetworksanddeeplearning.com/chap2.html

» Example: Data, Loss, Algorithm, and Metric

- **InClass demo: Implementation using tf.keras in Colab**
- Housing price dataset

$$\underset{\theta}{\operatorname{argmin}} \ \frac{1}{n} \sum_{i=1}^{n} L\big(y_i, \boldsymbol{f}(\boldsymbol{x}_i)\big)$$

- **Data.** Input features: $\boldsymbol{x}_i \in \mathbb{R}^d$; Output: $y_i \in \mathbb{R}$;
- **Model.** Predicting the house price: $\boldsymbol{f}(\boldsymbol{x}) \to y$;
- **Loss function.** RMSE or MSE;

$$L(y_i, f(\boldsymbol{x}_i)) = -\big(y_i - f(\boldsymbol{x}_i)\big)^2.$$

- **Evaluation metric.** MSE and RMSE

» Neural Networks: Cross-Validation

Step 1 Design your **neural network** with candidate hyperparameters

- param : weight matrix, intercept vector
- hps : depth, number of neurons, types of layers

Step 2 Train model parameters based on the training set with different hyperparameters

$$\widehat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L\big(y_i, \mathbf{f}_L(\mathbf{x}_i)\big)$$

Step 3 Compute validation loss for each hyperparameter setting using a validation set or k-fold cross-validation, and select the optimal architecture

Step 4 Refit the model with the optimal hps using all data

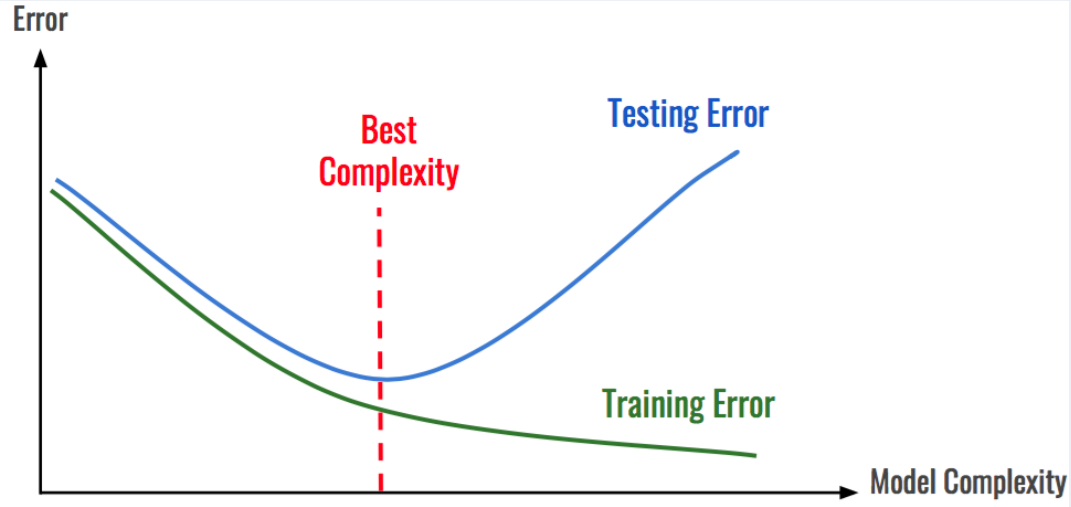
Step 5 Make predictions on the test set

» Neural Networks: Cross-Validation

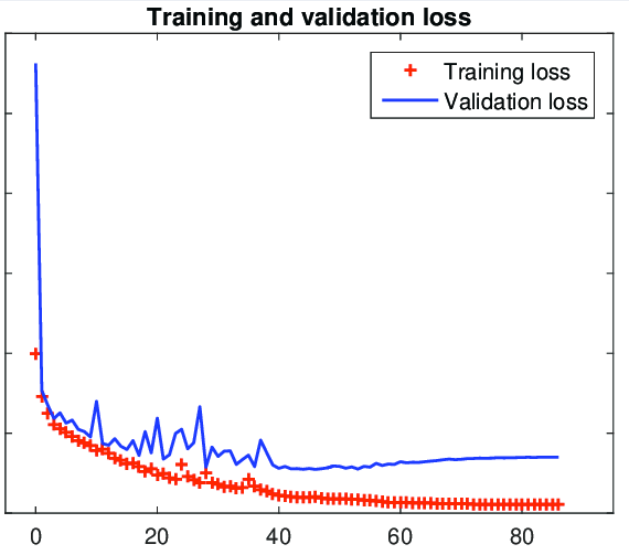
- Cross-validation (CV) in the [Previous Page] is entirely correct, but rarely used in practice for neural networks
- Training a neural network is not easy…
  - There are too many **hyperparameters (hp)**
  - For example, training a CNN on 16 vCPUs: **200 epochs took us 5 days to run**. [Source]
- Solution: Monitor the model's performance on a validation set and use early-stopping: stop training when a monitored validation metric has stopped improving

» Neural networks: bias-variance trade-off

ML: x-axis: Model complexity VS y-axis: Error



DL: x-axis: #iteration VS y-axis: Error



» Neural Networks: Early Stopping

If we can stop training before overfitting occurs ...
Monitoring and Early Stopping can be employed:

```
Epoch 1/30
112/112 - 6s - loss: 1.1414e-09 - accuracy: 1.0000
- validation loss: 2.6730e-10 - validation accuracy: 1.0000
...
Epoch 4/30
112/112 - 4s - loss: 1.0782e-10 - accuracy: 1.0000
- validation loss: 4.6980e-11 - validation accuracy: 1.0000
Epoch 5/30
112/112 - 4s - loss: 7.6734e-11 - accuracy: 1.0000
- validation loss: 3.4825e-11 - validation accuracy: 1.0000
Epoch 6/30
Restoring model weights from the best epoch: 1.
112/112 - 5s - loss: 5.8153e-11 - accuracy: 1.0000
- validation loss: 2.7316e-11 - validation accuracy: 1.0000
Epoch 6: early stopping
```

» Rules of Thumb: Neural Networks

Designing a NN can be overly flex, so here are some rules:

- Determine the problem type, and select the corresponding output layer activation function, loss function, and evaluation metric.
- Choose the number of nodes in hidden layers: typically, the first hidden layer should have approximately half the number of input features, with subsequent layers halving in size (e.g., 128, 64, 32, ...).
- Select an activation: ReLU is often a good choice.
- Determine the number of epochs: start with 20 to assess model convergence and accuracy. If minimal success is achieved, increase the number of epochs. Otherwise, consider 100 epochs and combine with CV techniques.
- Choose a batch size: select from a geometric progression of 2, starting with 16. For imbalanced datasets, consider larger values, such as 128.

» Appendix: Universal approximation theorem (Optinal)

Theorem (Universal approximation theorem)

For any function[a] $f : \mathbb{R}^d \to \mathbb{R}^K$ *and any* $\varepsilon > 0$*, there exists a fully-connected ReLU network* g *of width exactly* $\max(d+1, K)$*, such that*

$$\|f - g\|_p^p \leq \varepsilon.$$

[a] Bochner-Lebesgue p-integrable function

* Universal approximation theorem implies that a deep neural network can approximate an arbitrary function.

» Universal Approximation Theorem (Optional)

Example: Approximating the Sine Function with a Neural Network

- Ground Truth: $f^*(x) = \sin(x)$
- Network: a two-layer network with 100 neurons per layer
- Fitting results: Source