Neural Networks

STAT3009 Recommender Systems

by Ben Dai (CUHK)
on Department of Statistics and Data Science

» Today's Roadmap

1. What are neural networks? (Architecture)
   * Model structure, parameters vs. hyperparameters
2. How do we train them? (Gradient Descent + SGD)
   * Optimization, backpropagation
3. How do we implement them? (Keras)
   * Code examples, connecting math to implementation
4. How do we prevent overfitting? (Early Stopping)
   * Cross-validation, monitoring validation loss
5. Practical guidelines (Rules of Thumb)
   * Choosing hyperparameters, best practices

» Recall SVD Models

Recall the basic Latent Factor Model:

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

- 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

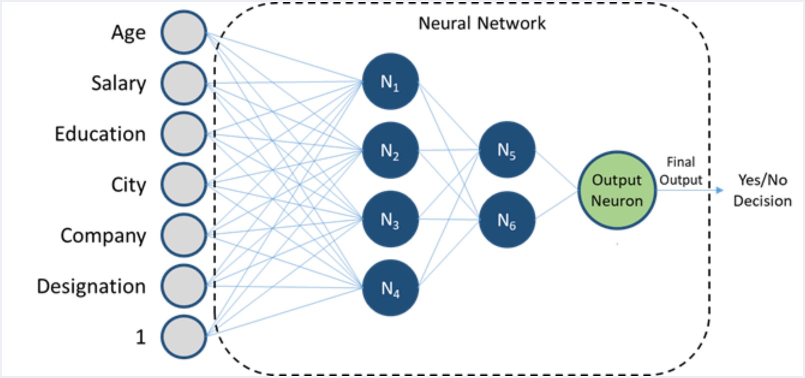
Q1 What are the parameters and hyperparameters for a neural network?

Q2 How do we train a neural network?

» Neural networks

Model architecture:

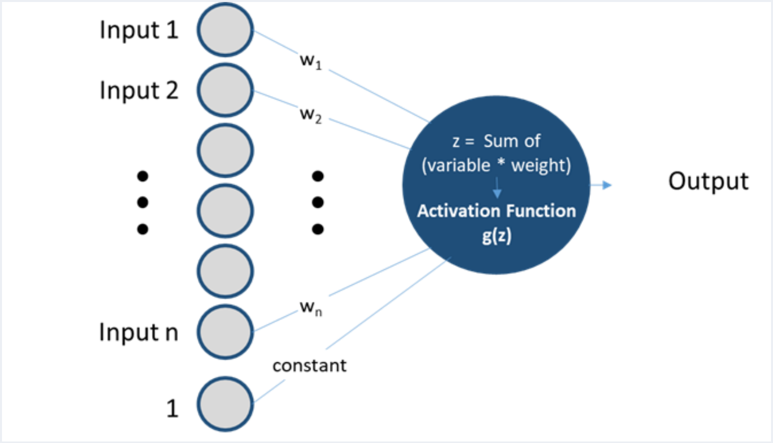
Input → Hidden Layer 1 → ⋯ → Hidden Layer L → Output



[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

» Neural networks

⚠ 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

» Neural networks: Parameters and Hyperparameters

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**: $\boldsymbol{b}_l \in \mathbb{R}^{d_l}$

» Neural networks: Parameters and Hyperparameters

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

» Neural networks: Training

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

General optimization problem:

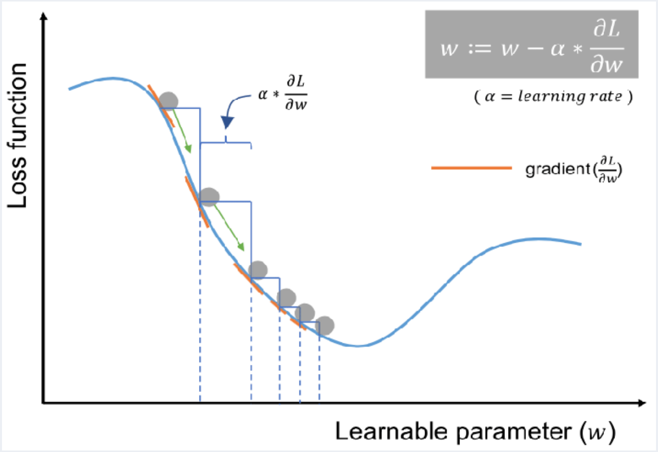
$$\min_{\theta} \mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L\big(y_i, \boldsymbol{f}_L(\boldsymbol{x}_i; \theta)\big)$$

* Gradient Descent: an iterative optimization method

  $$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_{\theta} \mathcal{L}(\theta^{(t)})$$

  where $\eta > 0$ is the learning rate (step size)
* Challenge: Computing the full gradient $\nabla_{\theta} \mathcal{L}(\theta)$ requires evaluating *all* n training samples $\rightarrow$ expensive!
* Solution: Use Stochastic Gradient Descent instead

» Gradient Descent: Visual Illustration



$w := w - \alpha * \frac{\partial L}{\partial w}$
$(\alpha = learning\, rate)$
$\alpha * \frac{\partial L}{\partial w}$
gradient$(\frac{\partial L}{\partial w})$
Loss function
Learnable parameter (w)

- Starting from an initial point
- At each step: $w := w - \alpha \frac{\partial L}{\partial w}$ (move opposite to gradient)
- Gradient (orange): slope of the loss at current point
- Step size controlled by learning rate α

» Example: Gradient Descent in Action

Problem: Minimize $L(\theta_1, \theta_2) = \theta_1^2 + 4\theta_2^2$ (2D quadratic)

Setup:

* Gradient:

$$\nabla L = \begin{pmatrix} 2\theta_1 \\ 8\theta_2 \end{pmatrix}$$

* Update:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla L(\theta^{(t)})$$

* Settings: $\eta = 0.2$, $\theta^{(0)} = (2, 1)$

Iterations:

$t = 0$: $\theta = (2.00, 1.00)$, $L = 8.00$

$t = 1$: $\theta = (1.20, 0.20)$, $L = 1.60$

$t = 2$: $\theta = (0.72, 0.04)$, $L = 0.52$

$t = 3$: $\theta = (0.43, 0.01)$, $L = 0.19$

$t = 4$: $\theta = (0.26, 0.00)$, $L = 0.07$

$\vdots$

$t \to \infty$: $\theta \to (0, 0)$, $L \to 0$

» Neural networks: Training

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

SGD Recall. Compute stochastic gradients for all model parameters

* Gradient:

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

* Approximation using one sample:

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

* Approximation using a mini-batch of samples

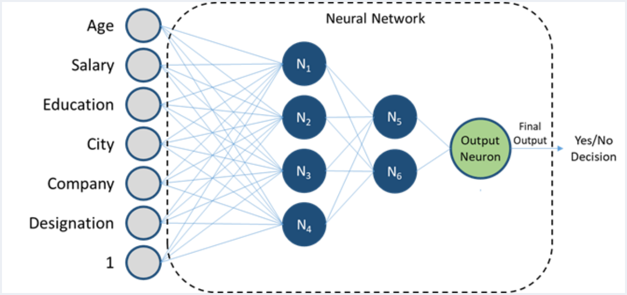
$$\frac{\partial \text{Loss}}{\partial \boldsymbol{\theta}} \leftarrow \frac{1}{|Batch|}\sum_{i \in Batch} \frac{\partial L(y_i, \mathbf{f}_L(\mathbf{x}_i))}{\partial \boldsymbol{\theta}}$$

» Neural networks: Training (Optional)

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

$$\begin{aligned}
&\text{Last layer} \quad \frac{\partial L(y_i,\mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{W}_L} = \frac{\partial L(y_i,\mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{f}_L(\mathbf{x}_i)} \frac{\partial \mathbf{f}_L(\mathbf{x}_i)}{\partial \mathbf{W}_L} \\
&\text{Layer L-1} \quad \frac{\partial L(y_i,\mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{W}_{L-1}} = \frac{\partial L(y_i,\mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{f}_L(\mathbf{x}_i)} \frac{\partial \mathbf{f}_L(\mathbf{x}_i)}{\partial \mathbf{f}_{L-1}(\mathbf{x}_i)} \frac{\partial \mathbf{f}_{L-1}(\mathbf{x}_i)}{\partial \mathbf{W}_{L-1}} \\
&\text{Layer L-2} \quad \frac{\partial L(y_i,\mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{W}_{L-2}} = \frac{\partial L(y_i,\mathbf{f}_L(\mathbf{x}_i))}{\partial \mathbf{f}_L(\mathbf{x}_i)} \frac{\partial \mathbf{f}_L(\mathbf{x}_i)}{\partial \mathbf{f}_{L-1}(\mathbf{x}_i)} \frac{\partial \mathbf{f}_{L-1}(\mathbf{x}_i)}{\partial \mathbf{f}_{L-2}(\mathbf{x}_i)} \frac{\partial \mathbf{f}_{L-2}(\mathbf{x}_i)}{\partial \mathbf{W}_{L-2}} \\
&\qquad \ldots
\end{aligned}$$

Application of the chain rule!

» Neural Networks: Training

Stochastic Gradient Descent (SGD) involves additional hyperparameters

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

* 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

» TensorFlow and Keras: Neural Networks

- **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** $\boldsymbol{f}(\boldsymbol{x})$
  - 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

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

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

» Keras Implementation

Connecting mathematics to code:

* Step 1: Define the model $\boldsymbol{f}_L(\boldsymbol{x};\theta)$
    * **model = tf.keras.Sequential([...])**
    * Specify layers, activation functions $\rightarrow$ architecture of $\boldsymbol{f}_L$
* Step 2: Compile the model - setup optimization
    * **model.compile(optimizer, loss, metrics)**
    * optimizer: SGD, Adam, etc. $\rightarrow$ algo to minimize $\mathcal{L}(\theta)$
    * loss: 'mse', etc. $\rightarrow L(y_i, \boldsymbol{f}_L(\boldsymbol{x}_i))$
    * metrics: 'accuracy', 'rmse', etc. $\rightarrow$ eval measures
* Step 3: Fit the model - solve optimization problem
    * **model.fit(X, y, epochs, batch_size, validation_data)**
    * epochs: number of passes through entire dataset
    * batch_size: size of mini-batch for SGD update
    * Solves: $\min_{\theta} \frac{1}{n}\sum_{i=1}^{n} L\big(y_i, \boldsymbol{f}_L(\boldsymbol{x}_i;\theta)\big)$

» 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

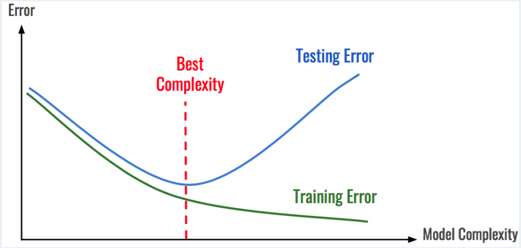
Step 5 Make predictions on the test set

» Neural Networks: Cross-Validation

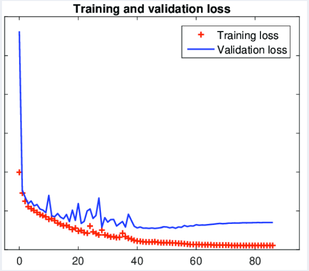
* Cross-validation (CV) in the [Previous Page] is entirely correct, but rarely used in practice for neural networks
* Training a neural network is not easy...
    * There are too many **hyperparameters (hp)**
    * For example, training a CNN on 16 vCPUs: **200 epochs took us 5 days to run**. [Source]
* 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



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

```
Epoch 1/50
loss: 0.4521 - accuracy: 0.7834
- val_loss: 0.3912 - val_accuracy: 0.8245
Epoch 2/50
loss: 0.3156 - accuracy: 0.8567
- val_loss: 0.2834 - val_accuracy: 0.8912
...
Epoch 8/50
loss: 0.0821 - accuracy: 0.9723
- val_loss: 0.1456 - val_accuracy: 0.9534  <- Best validation
Epoch 9/50
loss: 0.0634 - accuracy: 0.9812
- val_loss: 0.1523 - val_accuracy: 0.9501
Epoch 10/50
loss: 0.0512 - accuracy: 0.9856
- val_loss: 0.1689 - val_accuracy: 0.9478
Epoch 11/50
Restoring model weights from the end of the best epoch: 8.
Epoch 11: early stopping
```

» Keras: Early Stopping Implementation

Key arguments in tf.keras.callbacks.EarlyStopping:

```
early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True)

model.fit(X, y, validation_data=(X_val, y_val),
          epochs=50, callbacks=[early_stop])
```

- monitor: metric to track → typically 'val_loss' or 'val_accuracy'
  - Monitors validation performance to detect overfitting
- patience: number of epochs to wait before stopping
  - If monitored metric doesn't improve for **patience** epochs → stop
- restore_best_weights: whether to restore model weights from best epoch
  - If **True**: restores weights from epoch with best monitored metric

» Common Pitfalls to Avoid

Watch out for these mistakes when training NNs:

- Learning rate too high → Loss explodes or oscillates wildly
    - Symptoms: NaN losses, unstable training
- Learning rate too low → Training takes forever, gets stuck
    - Symptoms: Loss barely decreases after many epochs
- Forgetting to normalize inputs → Slow/unstable convergence
    - Solution: Standardize features to mean 0, std 1
- No validation set → Overfitting goes undetected
    - Monitor validation performance!
- Too many epochs without early stopping → Severe overfitting
- Extreme batch sizes: Batch size = 1 (too noisy) or = all data (too slow)

Solution: Start with reasonable defaults, then tune systematically!

» 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:
  - First hidden layer: $\approx$ half of input features
  - 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.

» Key Takeaways

What you should remember:

1. NNs = Universal approximators with layers of linear + nonlinear transforms
   * Architecture: $\boldsymbol{f}_l(\boldsymbol{x}) = A(\boldsymbol{W}_l \boldsymbol{f}_{l-1}(\boldsymbol{x}) + \boldsymbol{b}_l)$
2. Training = Optimization via gradient descent
   * Backpropagation computes gradients efficiently using chain rule
3. SGD trades accuracy for speed using mini-batches
   * Faster updates, can escape local minima
4. Hyperparameters matter: learning rate, batch size, architecture, epochs
5. Early stopping prevents overfitting by monitoring validation loss
6. Keras makes it easy: define $\rightarrow$ compile $\rightarrow$ fit

Next: Apply NNs to recommender systems (Neural Collaborative Filtering)!