Lecture 2: special layers, weights initialization, feature importance

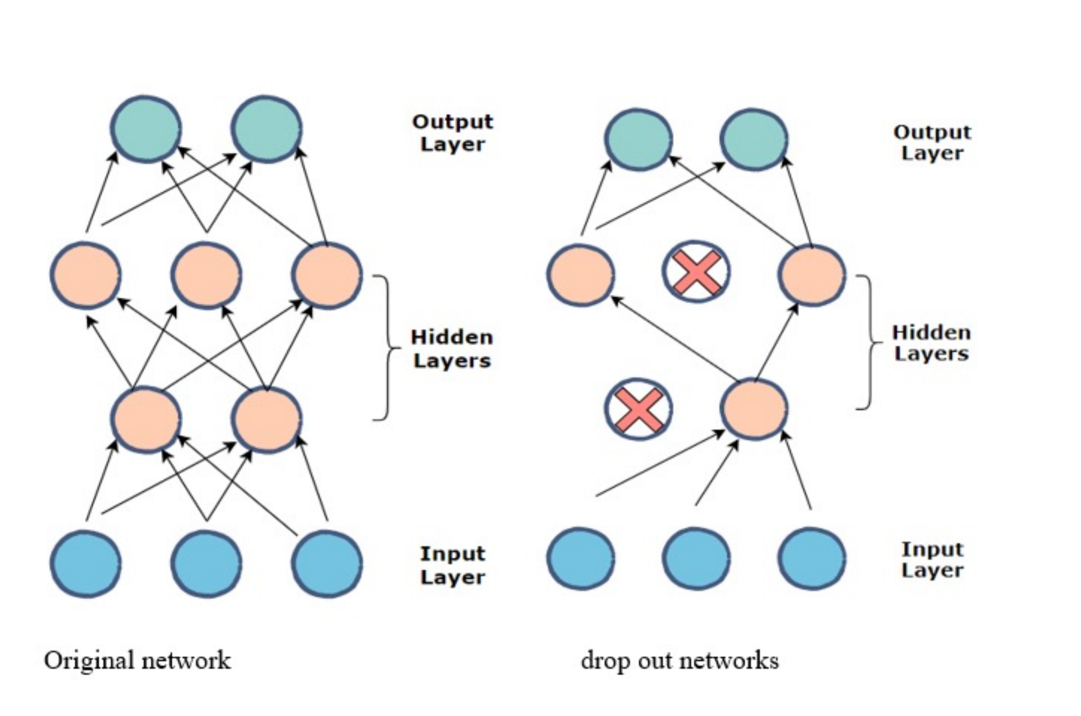
Machine Learning - 2

Data Science and Business Analytics Program (DSBA) at HSE & LSE, 22/23

Akim Tsvigun

Dropout

Idea: **prevent co-adaptation** of neurons from adjacent layers by **randomly zeroing** some of them **during training**



How does it look?

Input: activation function a, weights matrix W, input vector x, zeroing probability p

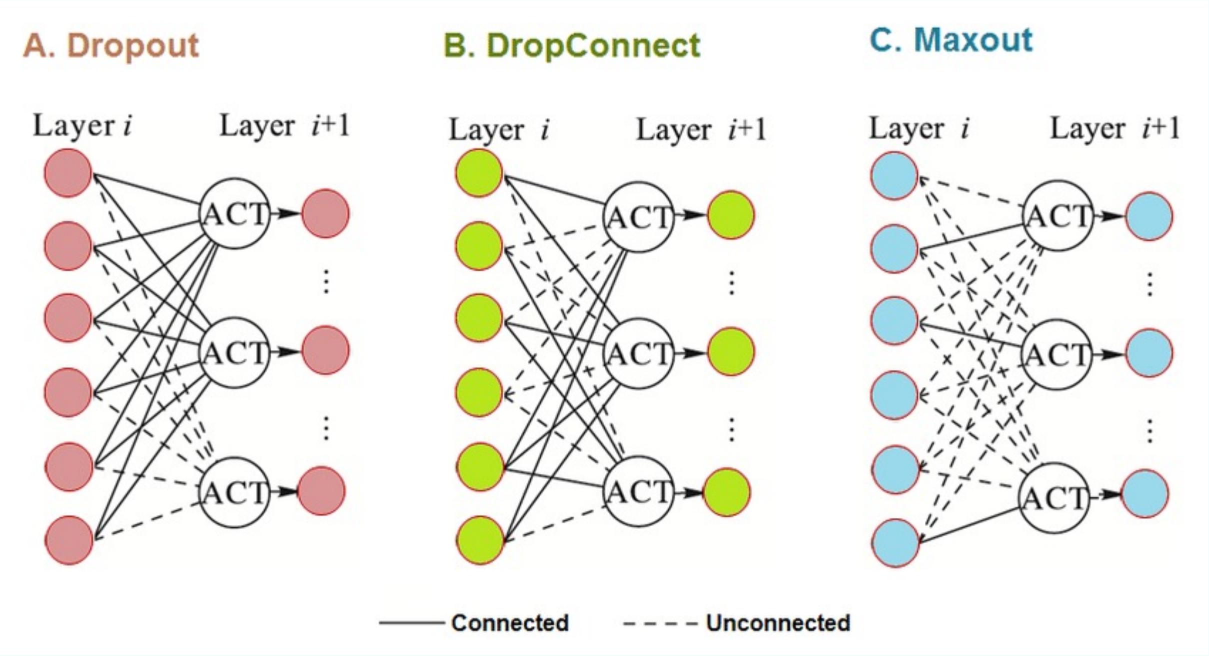
Output: $r = a((W \cdot x) \otimes M)$, where M is a binary matrix of the same shape as number of columns in W, $\mathrm{M_{ij}} \sim Bernoulli(p)$

Dropout Peculiarities

- Dropout is used to prevent **overfitting** of a NN
- **Re-scale** the output: divide the values from the **next** layer by (1 – p)
- A ’’default’’ value is 0.2
- Does not contain **learnable parameters**, one **hyperparameter**: p
- On inference, **disable** the dropout (`model.eval()` in Pytorch)
- Can use dropout for inference for uncertainty estimation (will see in Lecture 6)

Dropconnect

Idea: **prevent co-adaptation** of neurons from adjacent layers by **randomly zeroing** some of the **synapses (weights)**



How does it look?

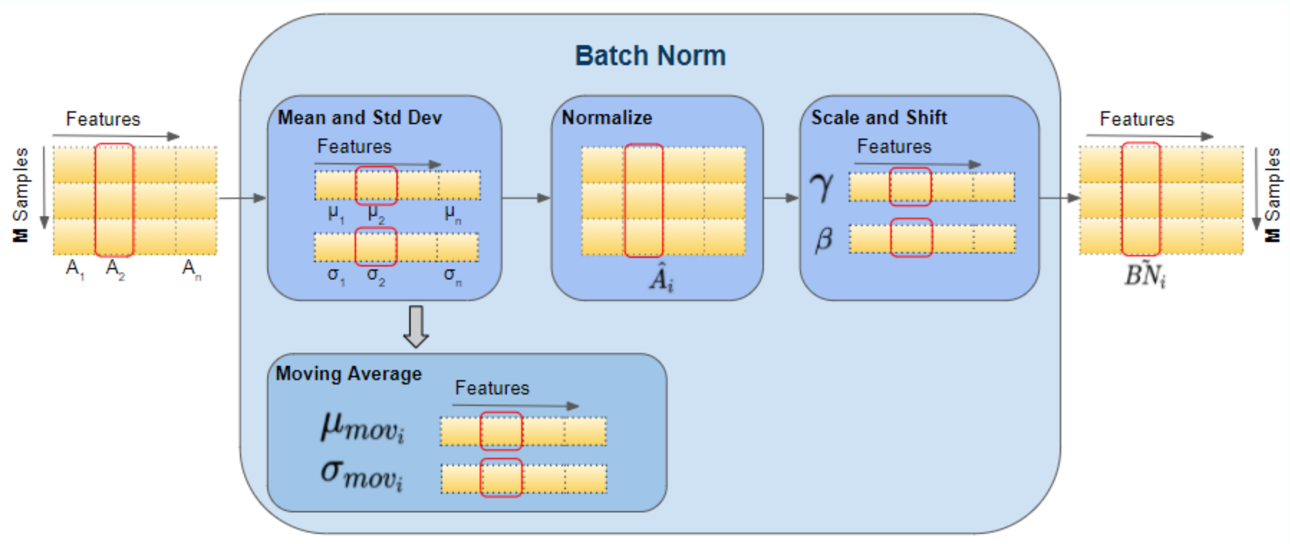
Input: activation function a, weights matrix W, input vector x, zeroing probability p

Output: $r = a((W \otimes M) \cdot x)$, where M is a binary matrix of the same shape as W, $\mathrm{M_{ij}} \sim Bernoulli(p)$

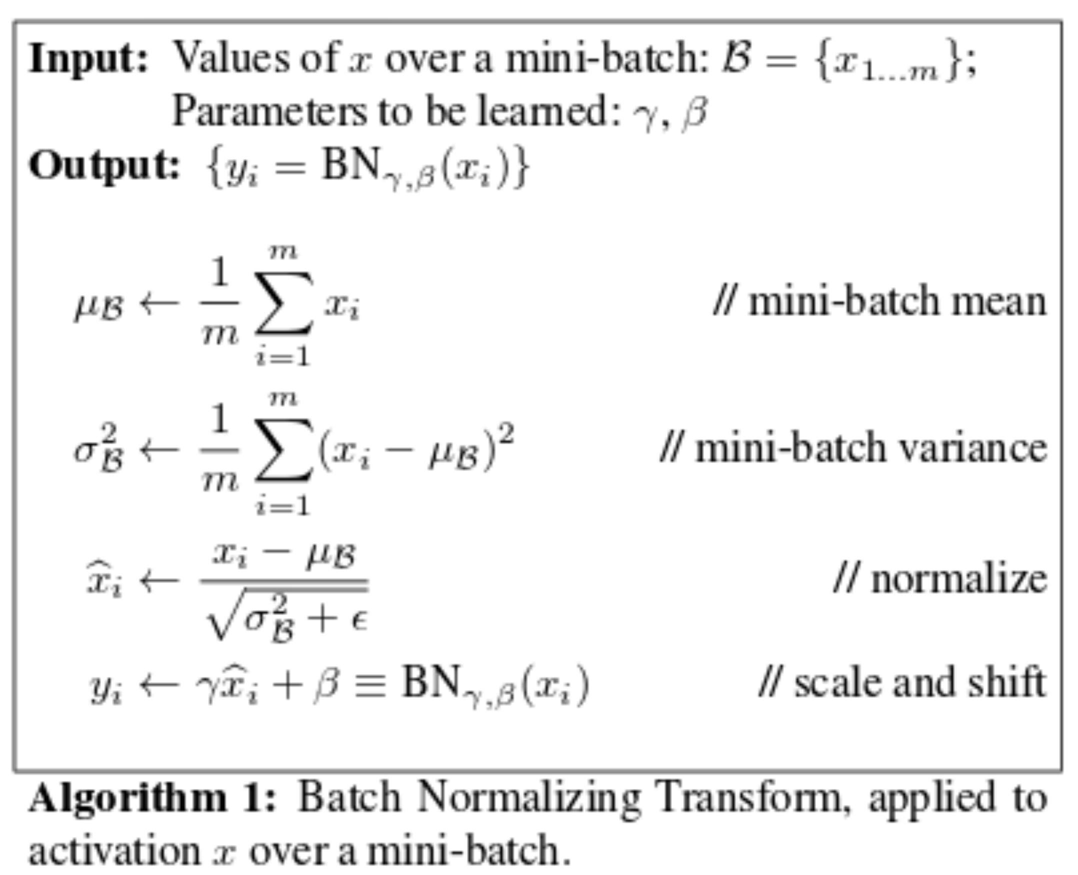
NATIONAL RESEARCH UNIVERSITY

Batch Normalization

Idea: **scale** each input feature w.r.t. feature values in the batch, and make a **linear transformation** to restore the scale



Formal algorithm



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

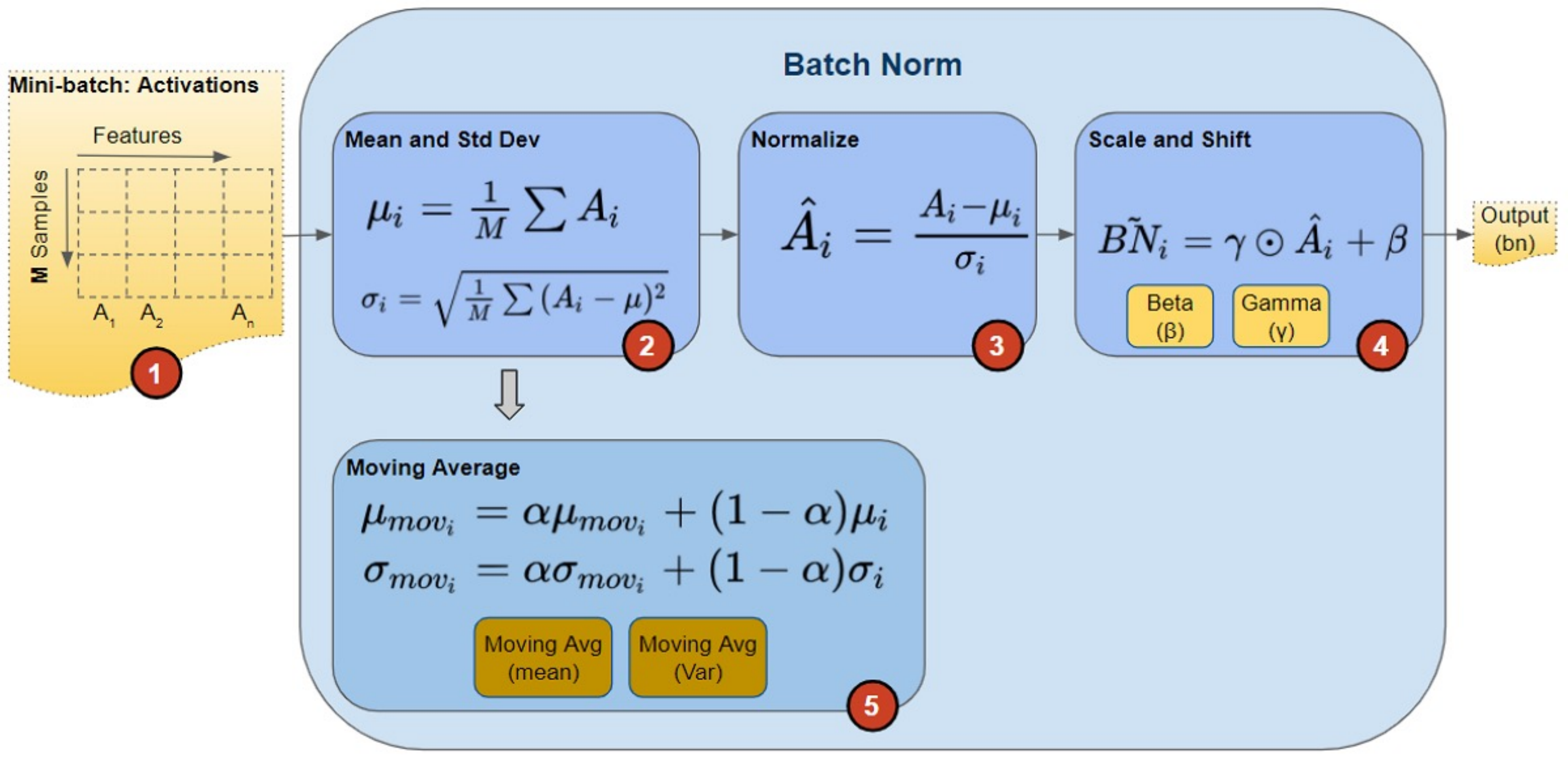
$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batch Normalization



Batch Norm

Mean and Std Dev

$$\mu_i = \frac{1}{M} \sum A_i$$

$$\sigma_i = \sqrt{\frac{1}{M} \sum (A_i - \mu)^2}$$

Normalize

$$\hat{A}_i = \frac{A_i - \mu_i}{\sigma_i}$$

Scale and Shift

$$B\tilde{N}_i = \gamma \odot \hat{A}_i + \beta$$

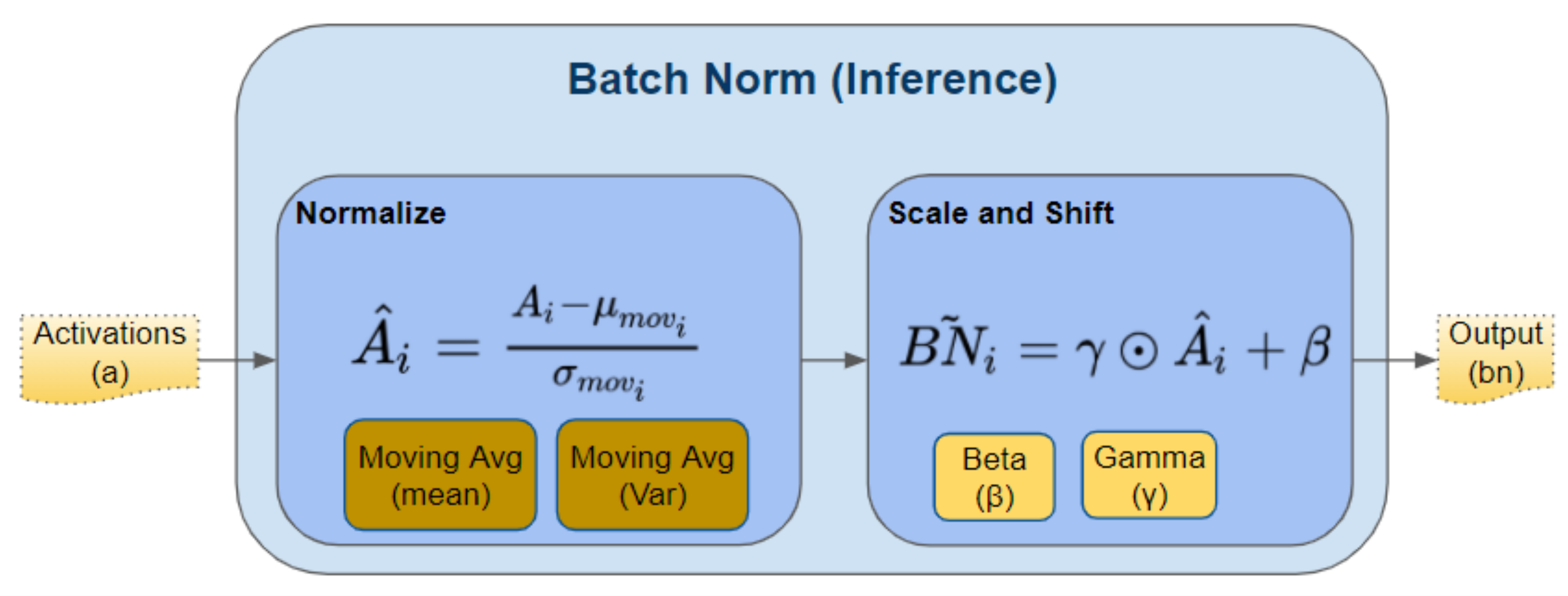
Moving Average

$$\mu_{mov_i} = \alpha \mu_{mov_i} + (1 - \alpha) \mu_i$$

$$\sigma_{mov_i} = \alpha \sigma_{mov_i} + (1 - \alpha) \sigma_i$$

Batch Normalization on Inference

Idea: inference should not depend on the batch → accumulate statistics during training



$$\hat{A}_i = \frac{A_i - \mu_{mov_i}}{\sigma_{mov_i}}$$

$$B\tilde{N}_i = \gamma \odot \hat{A}_i + \beta$$

Batch Normalization benefits

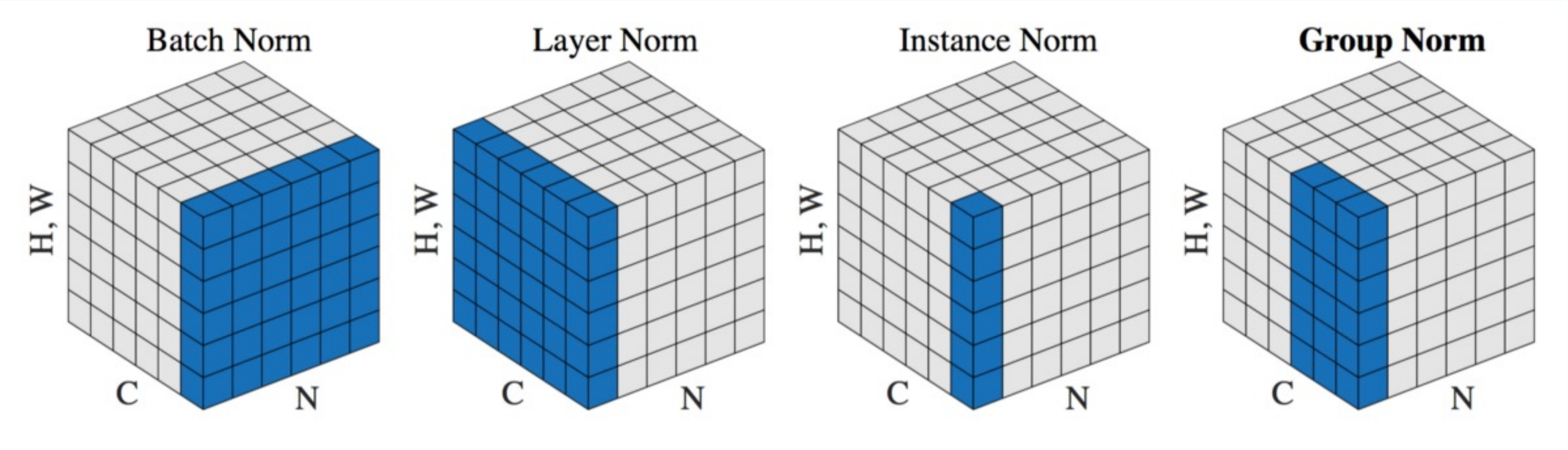
- Since we mostly use **assymetric** activation functions (ReLU-based), it ushers in many **positive values**. BN alleviates the problems by **renormalizing** the values
  - Scalar products of vectors of **different** objects will be **high**!
  - **Gradients of larger parameters** (corresponding to features with lower scale) will **dominate the update** for unscaled data!
- Many modern architectures use skip-connections. BN **normalizes the activation norm** and allows the skip-connections to **better dive into deep layers**
- BN **enhances the smoothness of the loss landscape**, allowing to use **larger LRs** (and hence train faster), especially for large batch sizes
- Some researchers believe that BN works as an **overfitting regularizer** since in-batch means and scales contain noise

Batch Normalization peculiarities

- On training, result for each element **depends** on the **other elements** in the batch
- **Large batch sizes** provide more **precise** in-batch statistics
- **Learnable parameters**: γ , β for each input feature (totally 2 * k parameters)
- **Hyperparameters**: α for statistics accumulation, ϵ for scaling
- During training, it **accumulates** the statistics in a momentum style
- On inference, BN uses the statistics **accumulated during training**

Layer Normalization

Idea: for 3 or more D tensors can scale across the second dimension!



Layer Normalization peculiarities

- **Does not depend** on the batch elements on training (in contrast to BN)
- **Does not accumulate** statistics
- **Learnable parameters**: γ , β for each input feature (totally 2 * k parameters)
- **Hyperparameters**: ϵ for scaling

NATIONAL RESEARCH UNIVERSITY

Order of Layers

Best practice: Linear → Norm → Activation → Dropout

Sometimes in CV is also used: Linear → Dropout → Activation → Norm

Weights Initialization

A constant value

(e.g. 0 or 1)

- Same gradients for different parameters with the same values → same values after update → parameters are just repeating each other!

A standard distribution with fixed std (e.g. $N(0;1)$ or $U(0;1)$)

- Variance and mean are not taken care of -> very large / small activation values -> exploding or vanishing gradient problem during backpropagation
- Aggravates with the depth increase

Weights Initialization

Solution: use distributions with tuned standard deviation!

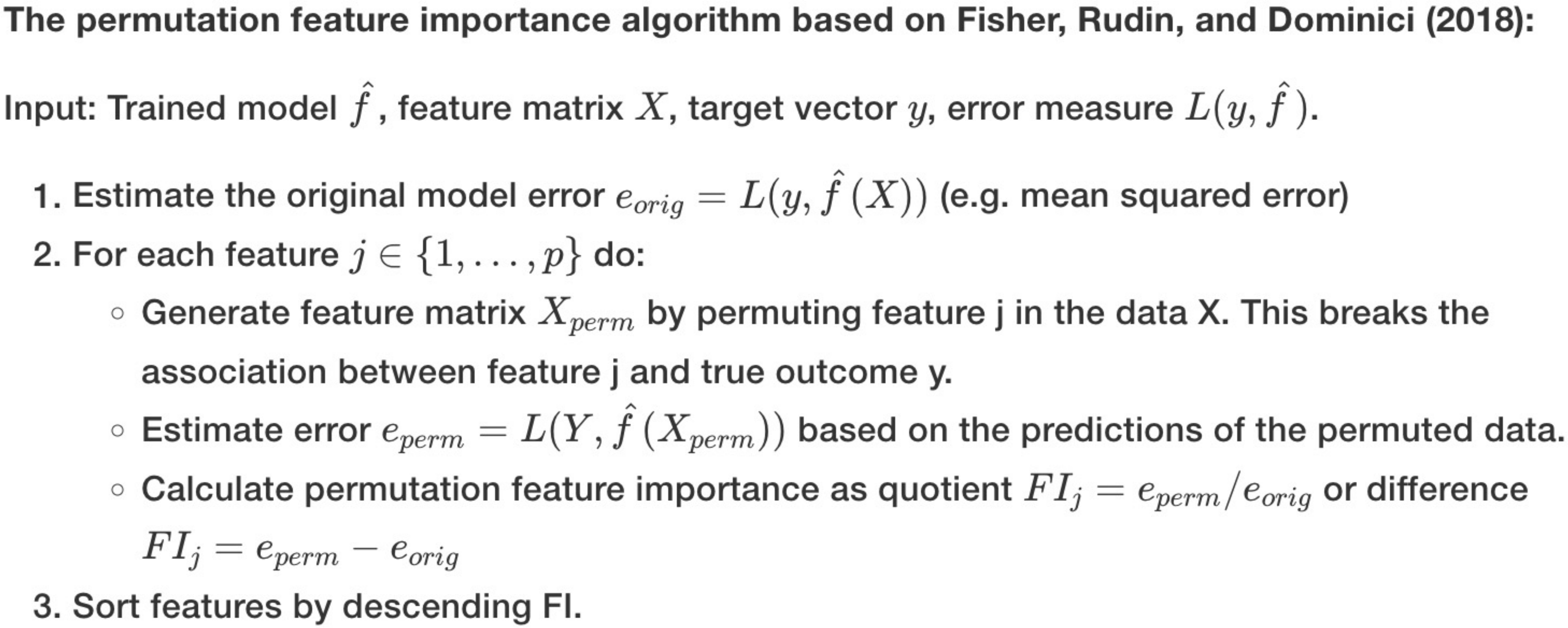
1. **Xavier (Glorot, He) Normal**:
   $w_i \sim N(0, \sigma^2);$
   $\sigma = gain \cdot \sqrt{\left(\frac{2}{f_{in}+f_{out}}\right)}$
2. **Xavier (Glorot, He) Uniform**:
   $w_i \sim U(-k, k);$
   $k = gain \cdot \sqrt{\left(\frac{6}{f_{in}+f_{out}}\right)}$

Theoretically proved to be a good choice when using **ReLU and its modifications** activations

1. **Kaiming Normal**:
   $w_i \sim N(0, \sigma^2); \ \ \sigma = \frac{gain}{\sqrt{f_{in}}}$
2. **Kaiming Uniform**:
   $w_i \sim U(-k, k); \ \ k = gain \cdot \sqrt{\frac{3}{f_{in}}}$

Theoretically proved to be a good choice when using **ReLU and its modifications** activations

Feature Importance



The permutation feature importance algorithm based on Fisher, Rudin, and Dominici (2018):

Input: Trained model $\hat{f}$, feature matrix X, target vector y, error measure $L(y, \hat{f})$.

1. Estimate the original model error $e_{orig} = L(y, \hat{f}(X))$ (e.g. mean squared error)
2. For each feature $j \in \{1, \ldots, p\}$ do:
   - Generate feature matrix X_{perm} by permuting feature j in the data X. This breaks the association between feature j and true outcome y.
   - Estimate error $e_{perm} = L(Y, \hat{f}(X_{perm}))$ based on the predictions of the permuted data.
   - Calculate permutation feature importance as quotient $FI_j = e_{perm}/e_{orig}$ or difference $FI_j = e_{perm} - e_{orig}$
3. Sort features by descending FI.

Feature importance

Which data split (i.e. train / eval) to use?

From Sklearn docs:

> *Permutation importances can be computed either on the training set or on a held-out testing or validation set. Using a held-out set makes it possible to highlight which features contribute the most to the generalization power of the inspected model. Features that are important on the training set but not on the held-out set might cause the model to overfit.*

Summary

- Dropout – to tackle overfitting
- BatchNorm / LayerNorm – for stable training
- The optimal choice for layers order: Linear → Norm → Activation → Dropout
- Weights should be initialized randomly from a distribution with a tuned standard deviation
- Feature importance can be calculated via Permutation FI algorithm

AIRI

airi.net