CPEN 455: Deep Learning

Lecture 3: Multilayer Perceptron

Renjie Liao

University of British Columbia

Winter, Term 1, 2026

Outline

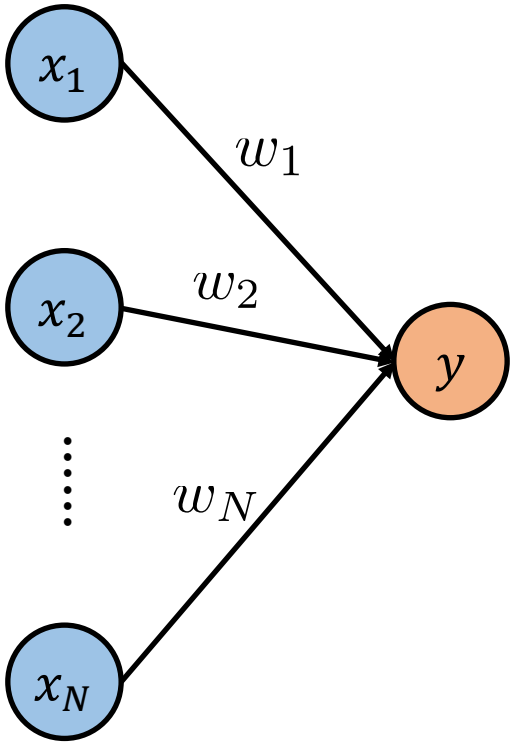
- Multilayer Perceptron (MLP)
  - Linear Layer
  - Activation Functions
  - Batch Normalization
  - Dropout
  - Build a Deep MLP

Outline

- Multilayer Perceptron (MLP)
  - **Linear Layer**
  - Activation Functions
  - Batch Normalization
  - Dropout
  - Build a Deep MLP

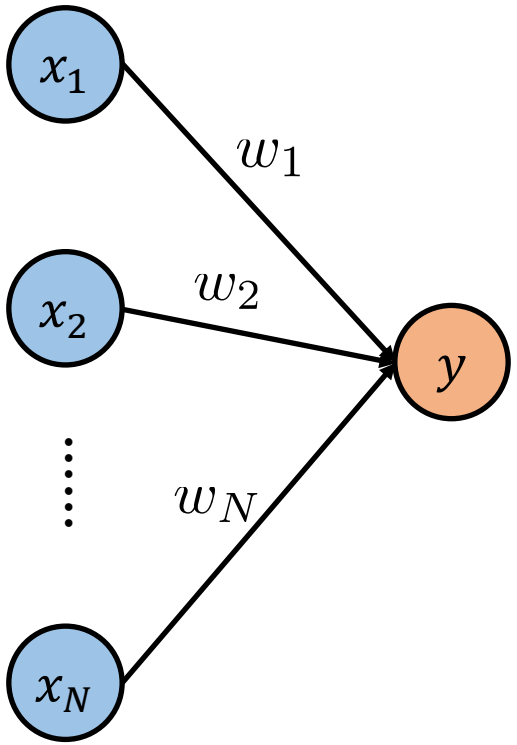
Linear Layer

Consider the following linear layer of a single output unit:



Linear Layer

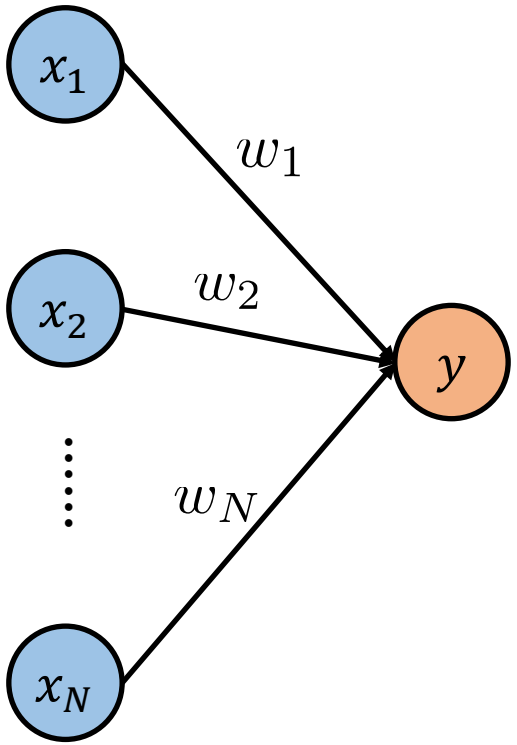
Consider the following linear layer of a single output unit:



$$y = \sum_{n=1}^{N} w_n x_n$$

Linear Layer

Consider the following linear layer of a single output unit:

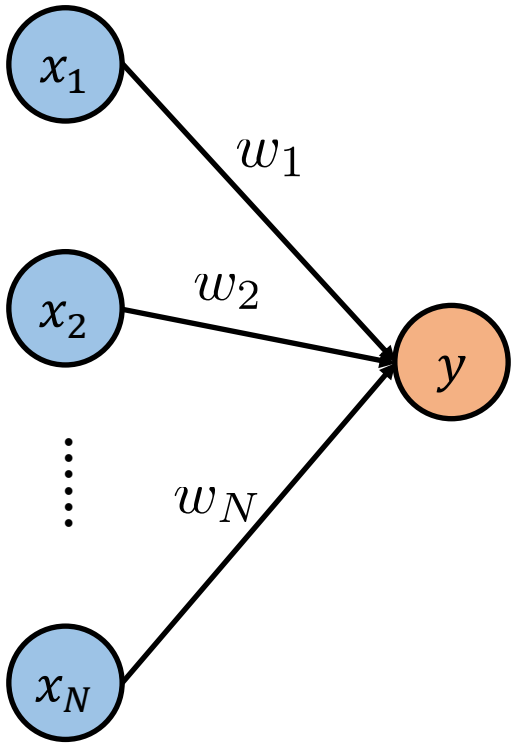


$$y = \sum_{n=1}^{N} w_n x_n$$

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}$$

Linear Layer

Consider the following linear layer of a single output unit:



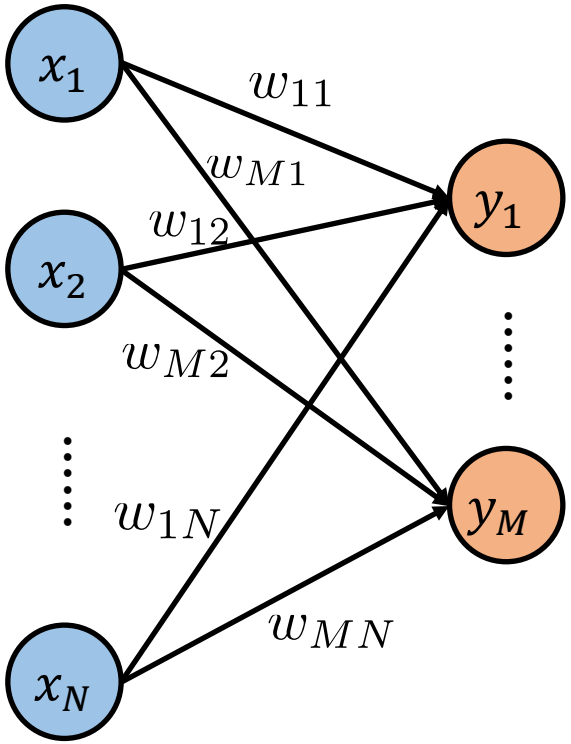
$$y = \sum_{n=1}^{N} w_n x_n = \mathbf{w}^\top \mathbf{x}$$

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}$$

Linear Layer

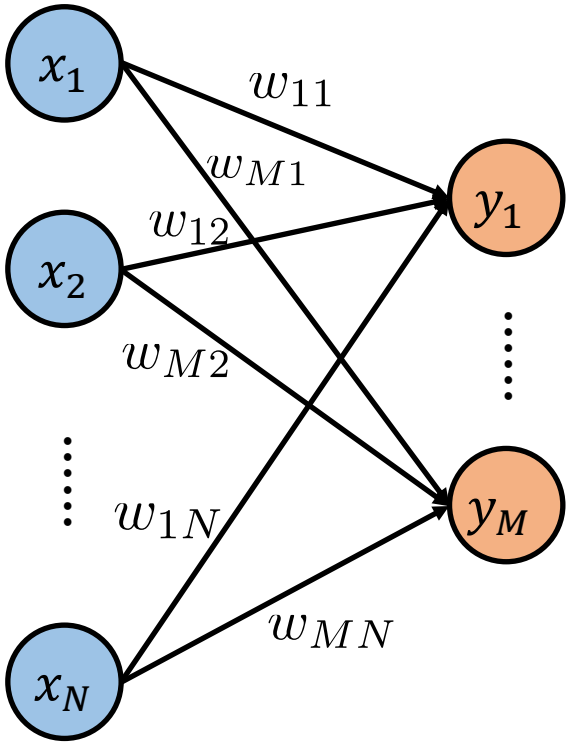
What if we have multiple output units?

$$y_m = \sum_{n=1}^{N} w_{mn} x_n$$



Linear Layer

What if we have multiple output units?

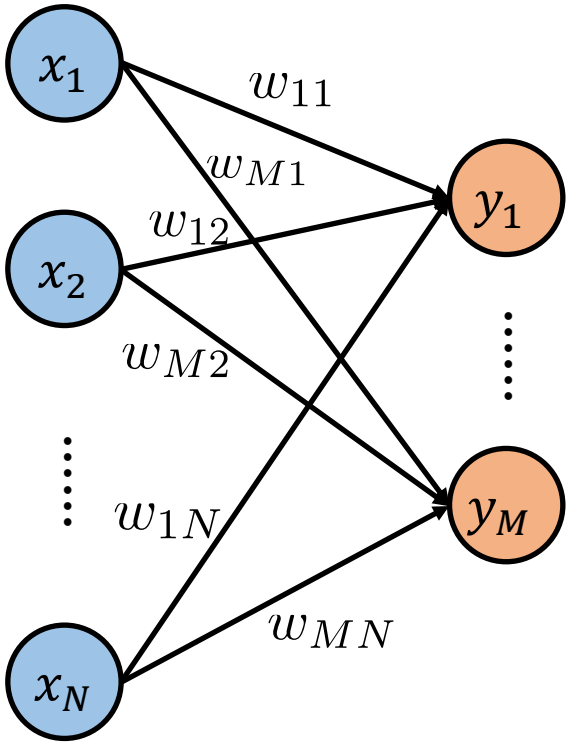


$$y_m = \sum_{n=1}^{N} w_{mn} x_n$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} \qquad W = \begin{bmatrix} w_{11}, & w_{12}, & \cdots, & w_{1N} \\ w_{21}, & w_{22}, & \cdots, & w_{2N} \\ \vdots, & \vdots, & \vdots, & \vdots \\ w_{M1}, & w_{M2}, & \cdots, & w_{MN} \end{bmatrix}$$

Linear Layer

What if we have multiple output units?



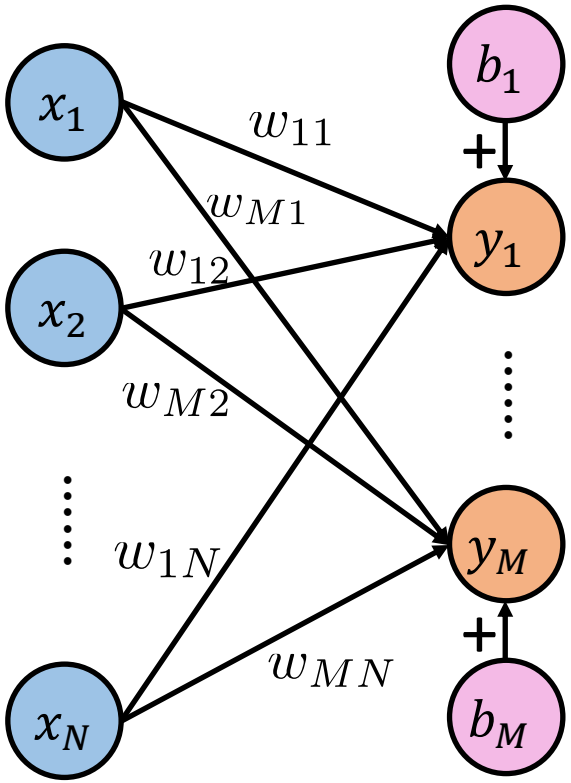
$$y_m = \sum_{n=1}^{N} w_{mn} x_n$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} \qquad W = \begin{bmatrix} w_{11}, & w_{12}, & \cdots, & w_{1N} \\ w_{21}, & w_{22}, & \cdots, & w_{2N} \\ \vdots, & \vdots, & \vdots, & \vdots \\ w_{M1}, & w_{M2}, & \cdots, & w_{MN} \end{bmatrix}$$

$$\mathbf{y} = W\mathbf{x}$$

Linear Layer

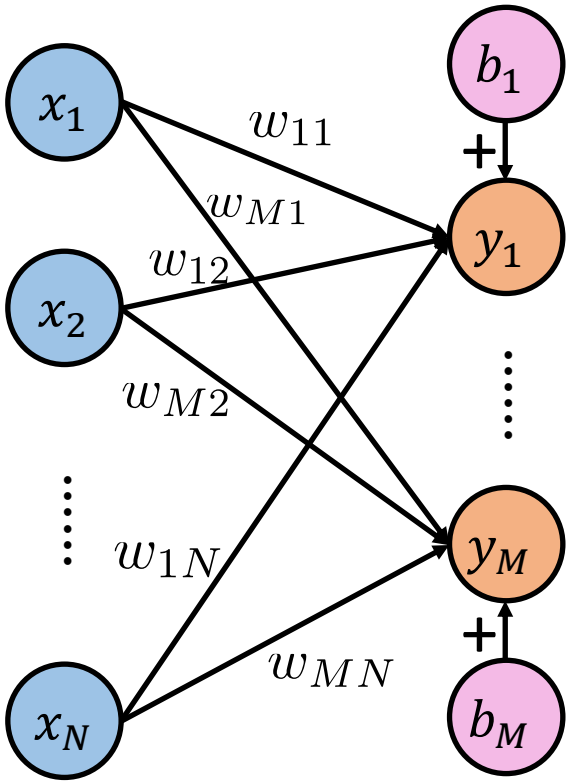
What if we have multiple output units? What about bias?



$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_M \end{bmatrix}$$

Linear Layer

What if we have multiple output units? What about bias?



$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_M \end{bmatrix}$$

$$\mathbf{y} = W\mathbf{x} + \mathbf{b}$$

Linear Layer

A linear layer with a bias is an affine map.

Stacking affine layers alone still gives one affine map.

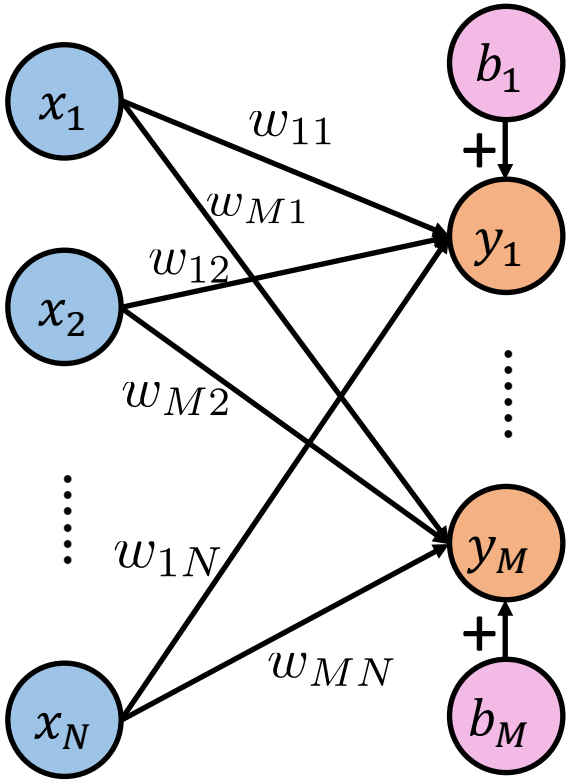
$$W_2(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (W_2W_1)\mathbf{x} + (W_2\mathbf{b}_1 + \mathbf{b}_2)$$

Nonlinear activations let depth represent more complex functions.

Linear Layer

What if we have multiple output units? What about bias?

We can compactly rewrite it via homogeneous coordinates.

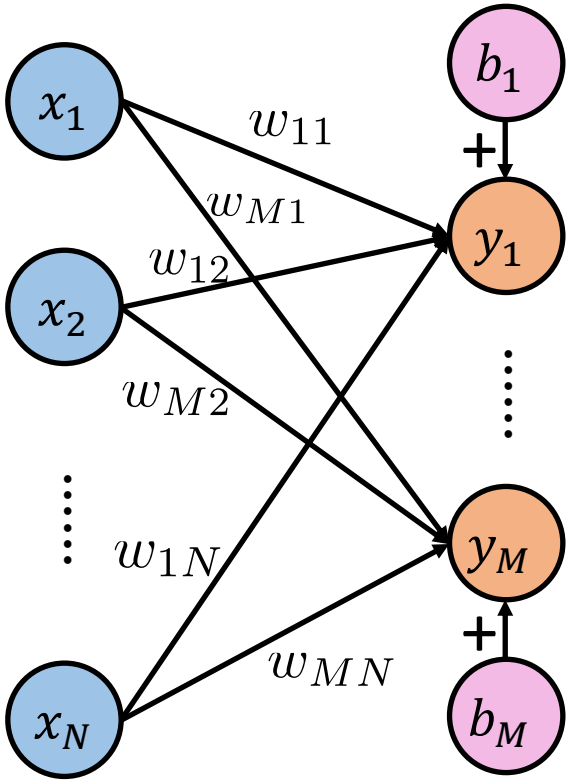


$$\tilde{\mathbf{x}} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \\ 1 \end{bmatrix}$$

Linear Layer

What if we have multiple output units? What about bias?

We can compactly rewrite it via homogeneous coordinates.

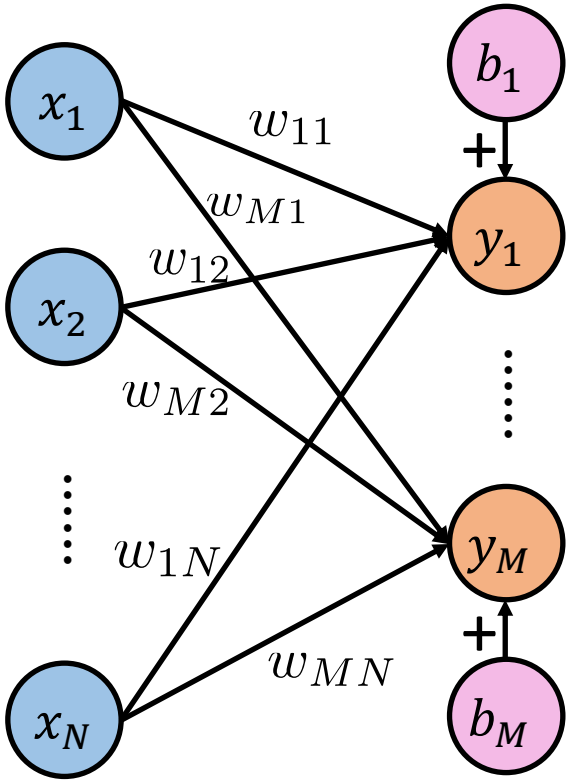


$$\tilde{\mathbf{x}} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \\ 1 \end{bmatrix} \quad \tilde{W} = \begin{bmatrix} w_{11}, & w_{12}, & \cdots, & w_{1N} & b_1 \\ w_{21}, & w_{22}, & \cdots, & w_{2N} & b_2 \\ \vdots, & \vdots, & \vdots, & \vdots & \vdots \\ w_{M1}, & w_{M2}, & \cdots, & w_{MN} & b_M \end{bmatrix}$$

Linear Layer

What if we have multiple output units? What about bias?

We can compactly rewrite it via homogeneous coordinates.



$$\tilde{\mathbf{x}} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \\ 1 \end{bmatrix} \quad \tilde{W} = \begin{bmatrix} w_{11}, & w_{12}, & \cdots, & w_{1N} & b_1 \\ w_{21}, & w_{22}, & \cdots, & w_{2N} & b_2 \\ \vdots, & \vdots, & \vdots, & \vdots & \vdots \\ w_{M1}, & w_{M2}, & \cdots, & w_{MN} & b_M \end{bmatrix}$$

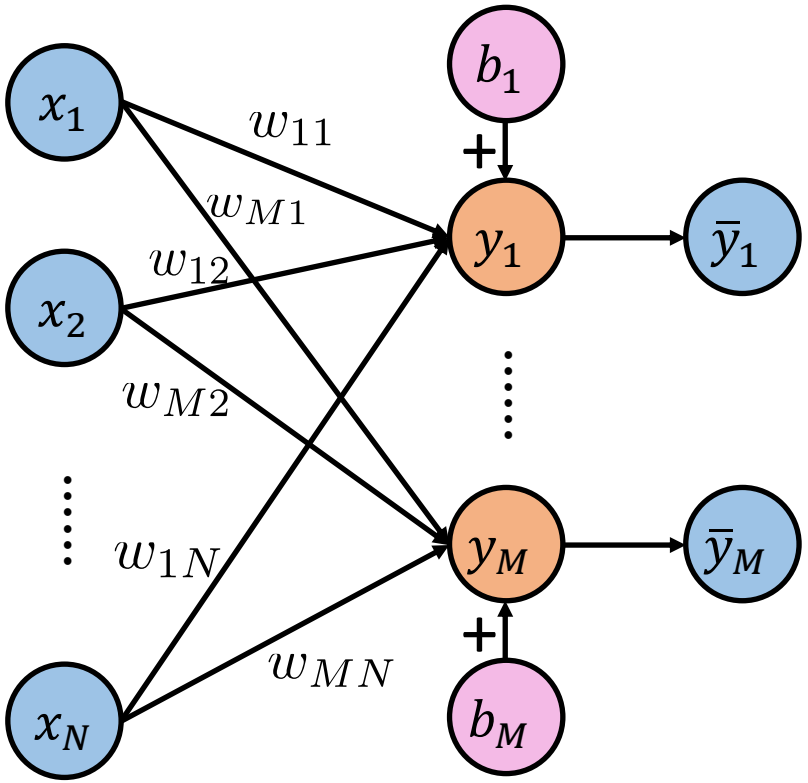
$$\mathbf{y} = W\mathbf{x} + \mathbf{b} = \tilde{W}\tilde{\mathbf{x}}$$

Outline

- Multilayer Perceptron (MLP)
  - Linear Layer
  - **Activation Functions**
  - Batch Normalization
  - Dropout
  - Build a Deep MLP

Activation Functions

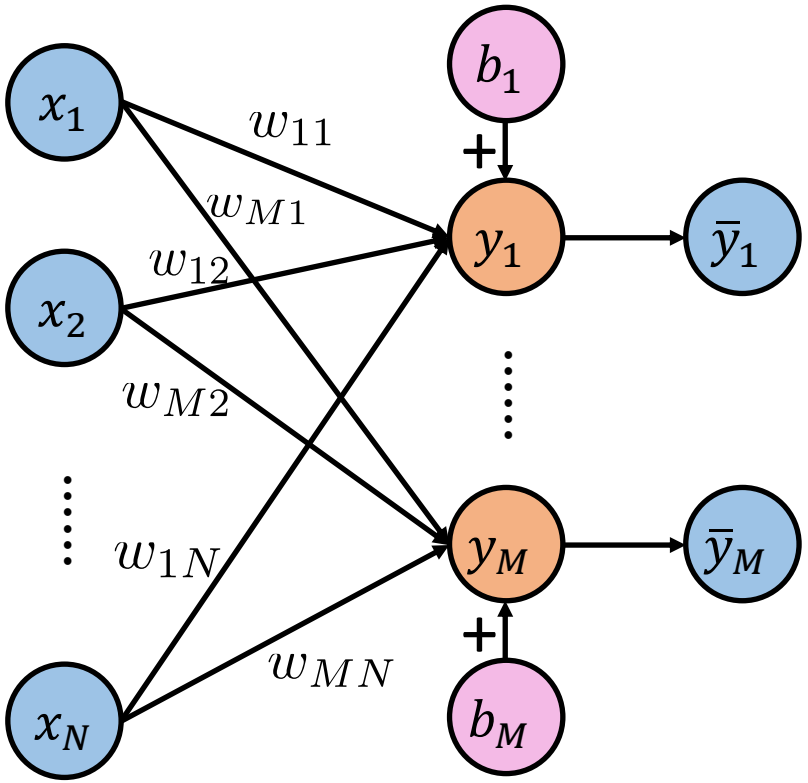
To make neural networks become nonlinear models, we often apply **element-wise** nonlinear activation functions.



$$\bar{y}_i = f(y_i)$$

Activation Functions

To make neural networks become nonlinear models, we often apply **element-wise** nonlinear activation functions.



$$\bar{y}_i = f(y_i)$$

$$\bar{\mathbf{y}} = f(\mathbf{y})$$

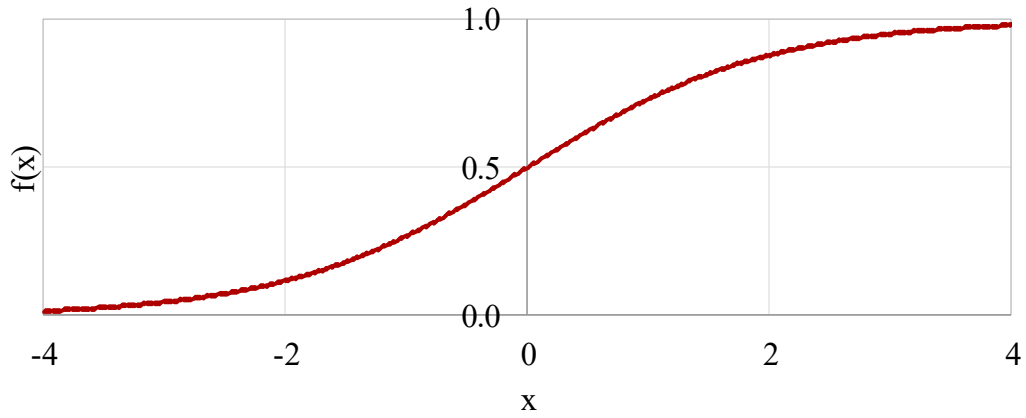
Activations act element-wise unless stated otherwise.

Activation Functions

We usually apply the activation independently to each coordinate.

- **Sigmoid**

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$



Useful for binary probabilities and gates.

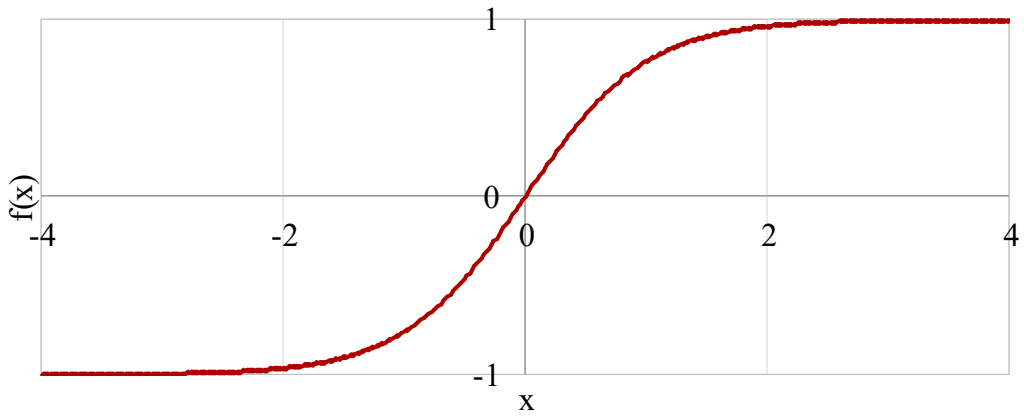
Saturates for large positive or negative inputs.

Activation Functions

We usually apply the activation independently to each coordinate.

- Sigmoid
- **Tanh**

$$f(x) = \tanh(x)$$



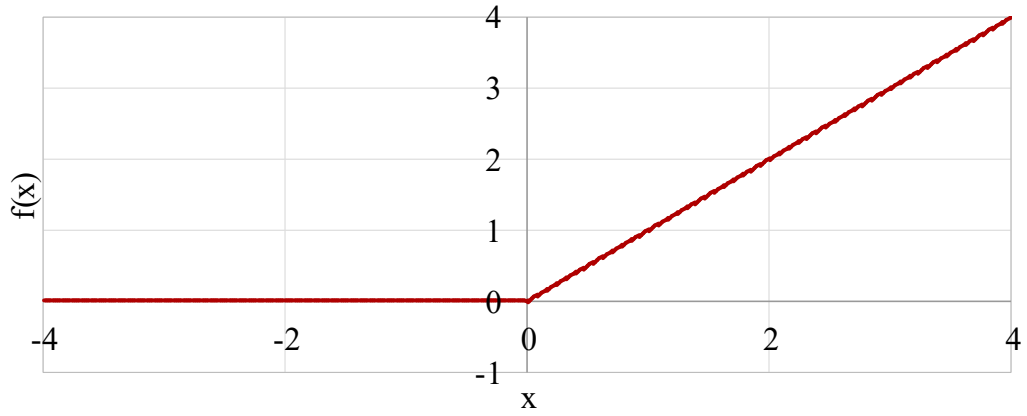
Zero-centered. Used in recurrent state updates.
Saturates on both sides.

Activation Functions

We usually apply the activation independently to each coordinate.

- Sigmoid
- Tanh
- **ReLU [1]**

$$f(x) = \max(0, x)$$



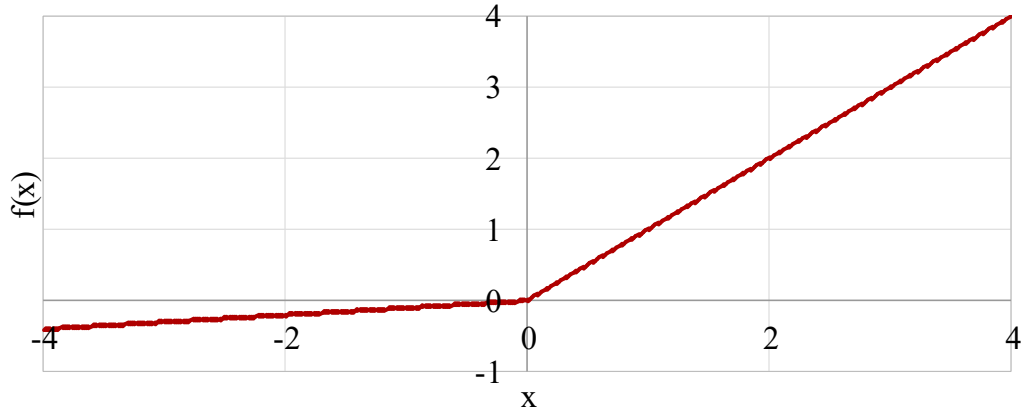
Zeroes negative inputs, giving sparse activations.
Zero gradient can leave units inactive.

Activation Functions

We usually apply the activation independently to each coordinate.

- Sigmoid
- Tanh
- ReLU [1]
- **Leaky ReLU / PReLU [2]**

$$f(x) = \begin{cases} x, & x \geq 0 \\ ax, & x < 0 \end{cases}$$



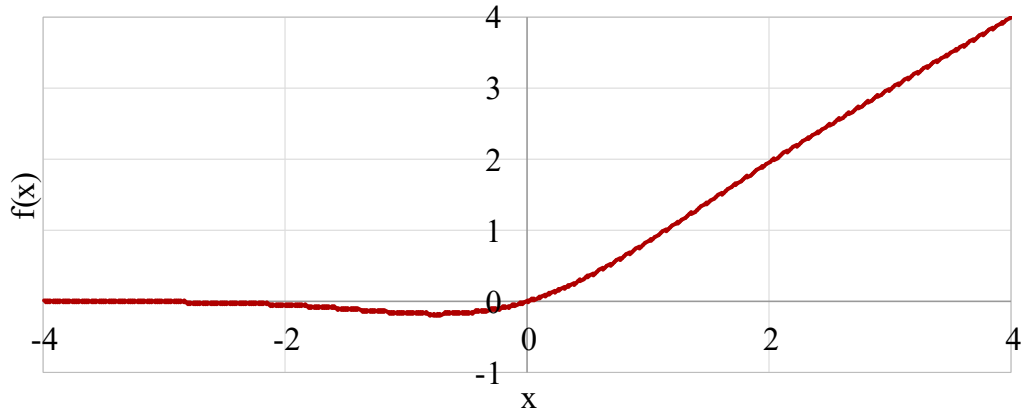
Leaky ReLU fixes the negative slope a.
PReLU learns a. The plot uses a = 0.1.

Activation Functions

We usually apply the activation independently to each coordinate.

- Sigmoid
- Tanh
- ReLU [1]
- Leaky ReLU / PReLU [2]
- **GELU [3]**

$$f(x) = x\Phi(x)$$



Φ is the standard normal CDF.

Smooth activation used in BERT and ViT.

Activation Functions

We usually apply the activation independently to each coordinate.

- Sigmoid
- Tanh
- ReLU [1]
- Leaky ReLU / PReLU [2]
- GELU [3]
- **SiLU / Swish [3, 4]**

$$f(x) = x\sigma(x)$$

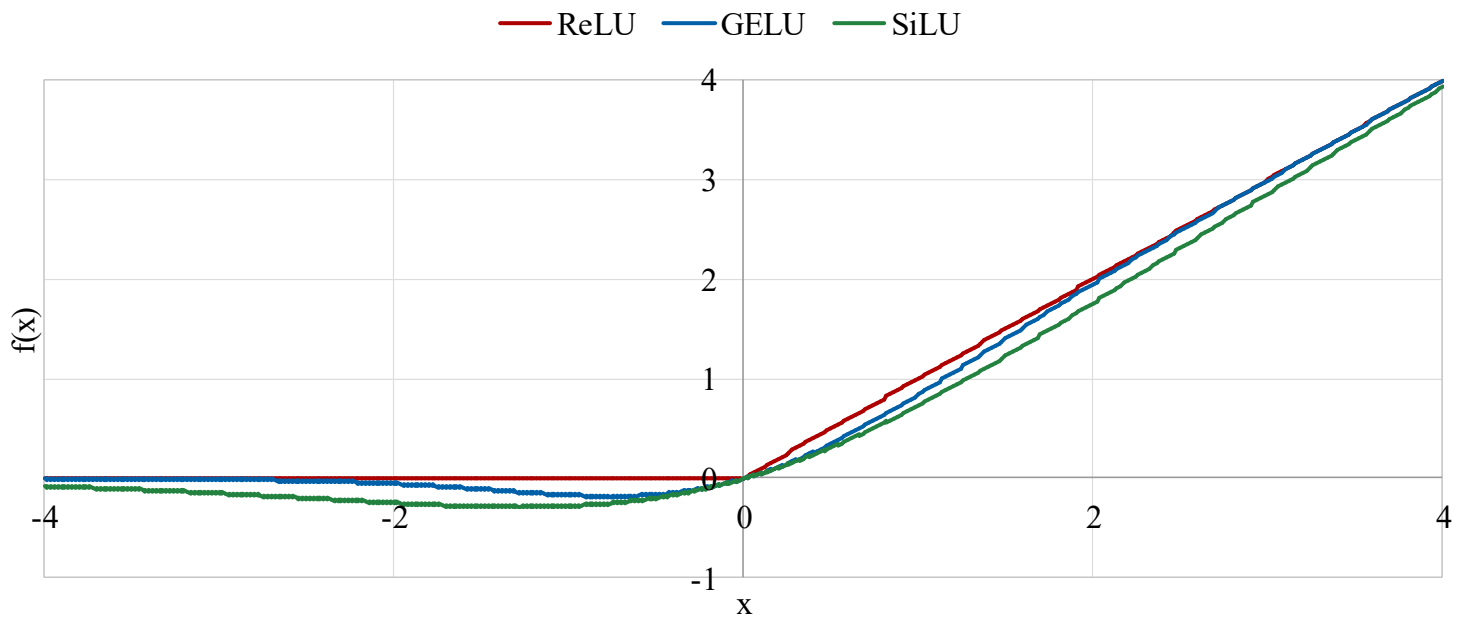
SiLU is Swish with β = 1.

Smooth self-gating. Used in EfficientNet and in SwiGLU gates.

Activation Functions

ReLU, GELU, and SiLU

GELU and SiLU keep small negative values and change smoothly near zero.



Choose the activation with the architecture. There is no universal winner.

Activation Functions

Some nonlinear functions couple multiple coordinates.

- Softmax

$$f(\mathbf{x}) = \left[\frac{\exp(\mathbf{x}_1)}{\sum_{k=1}^{K} \exp(\mathbf{x}_k)}, \ldots, \frac{\exp(\mathbf{x}_K)}{\sum_{k=1}^{K} \exp(\mathbf{x}_k)} \right]$$

- Maxout [9]

$$f(\mathbf{x}) = \max_i \quad \mathbf{x}_i$$

- cumax [10]

$$f(\mathbf{x}) = \left[\frac{\exp(\mathbf{x}_1)}{\sum_{k=1}^{K} \exp(\mathbf{x}_k)}, \ldots, \frac{\sum_{i=1}^{j} \exp(\mathbf{x}_i)}{\sum_{k=1}^{K} \exp(\mathbf{x}_k)}, \ldots, 1 \right]$$

Softmax followed by cumulative sum

Activation Functions

Gated MLPs [5]

Compute two learned projections, then multiply them element-wise.

$$\mathbf{u} = W_u\mathbf{h} + \mathbf{b}_u, \qquad \mathbf{v} = W_v\mathbf{h} + \mathbf{b}_v$$

$$\mathrm{GLU}(\mathbf{h}) = \sigma(\mathbf{u}) \odot \mathbf{v}$$

Activation Functions

Gated MLPs [5]

Compute two learned projections, then multiply them element-wise.

$$\mathbf{u} = W_u\mathbf{h} + \mathbf{b}_u, \qquad \mathbf{v} = W_v\mathbf{h} + \mathbf{b}_v$$

$$\mathrm{GLU}(\mathbf{h}) = \sigma(\mathbf{u}) \odot \mathbf{v}$$

GEGLU [5] uses GELU. SwiGLU [5] uses SiLU.

$$\mathrm{SwiGLU}(\mathbf{h}) = \mathrm{SiLU}(\mathbf{u}) \odot \mathbf{v}$$

At fixed hidden width, gating adds parameters. We return to this in Transformers.

Activation Functions

Some nonlinear functions couple multiple coordinates.

- Softmax

$$f(\mathbf{x}) = \left[\frac{\exp(\mathbf{x}_1)}{\sum_{k=1}^{K} \exp(\mathbf{x}_k)}, \ldots, \frac{\exp(\mathbf{x}_K)}{\sum_{k=1}^{K} \exp(\mathbf{x}_k)} \right]$$

- Maxout [9]

$$f(\mathbf{x}) = \max_i \quad \mathbf{x}_i$$

- cumax [10]

$$f(\mathbf{x}) = \left[\frac{\exp(\mathbf{x}_1)}{\sum_{k=1}^{K} \exp(\mathbf{x}_k)}, \ldots, \frac{\sum_{i=1}^{j} \exp(\mathbf{x}_i)}{\sum_{k=1}^{K} \exp(\mathbf{x}_k)}, \ldots, 1 \right]$$

Softmax followed by cumulative sum

Outline

- Multilayer Perceptron (MLP)
  - Linear Layer
  - Activation Functions
  - **Batch Normalization**
  - Dropout
  - Build a Deep MLP

Batch Normalization

Batch Normalization (BN) [6]

Normalize each feature using statistics across the mini-batch.

Batch Normalization

Batch Normalization (BN) [6]

Normalize each feature using statistics across the mini-batch.

Why use it?

It often improves optimization and allows larger learning rates.

Batch Normalization

Batch Normalization (BN) [6]

Normalize each feature using statistics across the mini-batch.

Why use it?

It often improves optimization and allows larger learning rates.

Original motivation: internal covariate shift

Reducing distribution shifts alone does not fully explain its benefits [7].

Its effect depends on optimization, batch statistics, and the model.

Batch Normalization

Suppose we have a batch of activations $X \in \mathbb{R}^{B \times D}$

vectorize

Batch Normalization

Suppose we have a batch of activations $X \in \mathbb{R}^{B \times D}$

vectorize

vectorize

vectorize

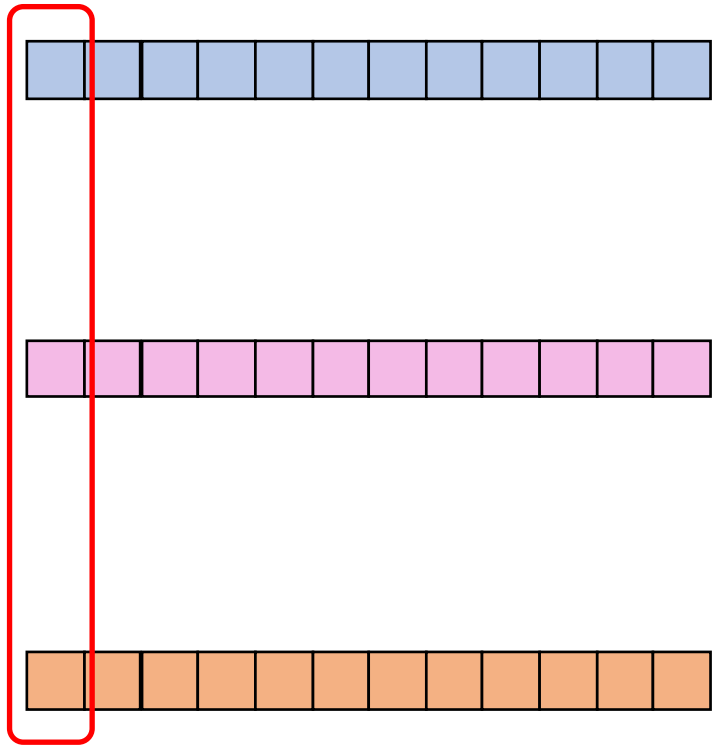
Batch Normalization

Suppose we have a batch of activations $X \in \mathbb{R}^{B \times D}$

vectorize

vectorize

vectorize



$$\mathbf{m}[j] = \frac{1}{B}\sum_{i=1}^{B} X[i,j]$$

$$\mathbf{m} = \frac{1}{B}\sum_{i=1}^{B} X[i,:]$$

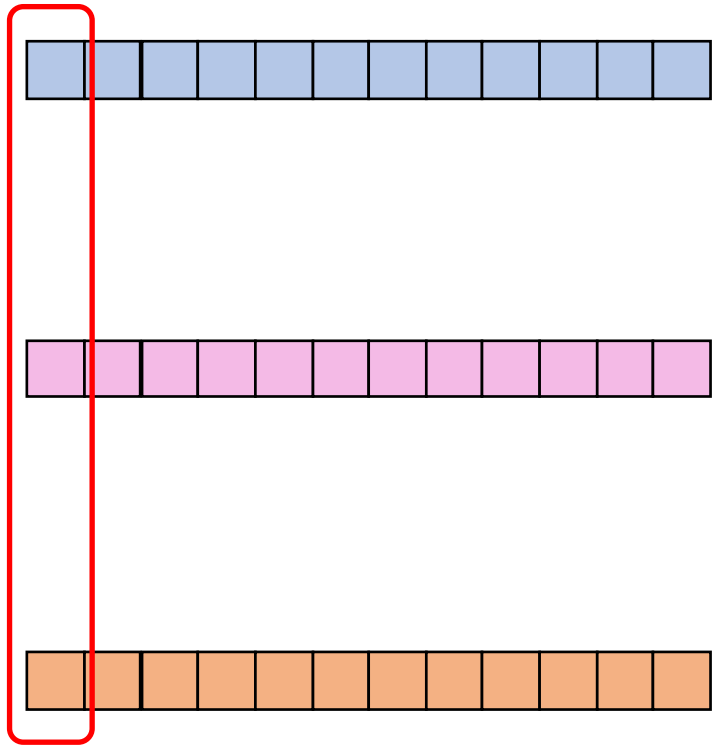
Batch Normalization

Suppose we have a batch of activations $X \in \mathbb{R}^{B \times D}$

vectorize

vectorize

vectorize



$$\mathbf{m}[j] = \frac{1}{B}\sum_{i=1}^{B} X[i,j]$$

$$\mathbf{v}[j] = \frac{1}{B}\sum_{i=1}^{B} (X[i,j] - \mathbf{m}[j])^2$$

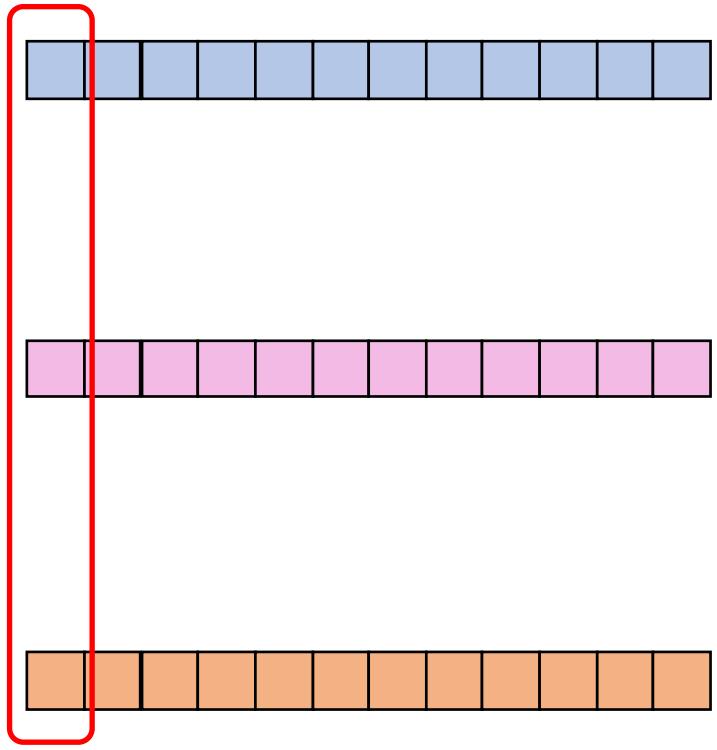
Batch Normalization

Suppose we have a batch of activations $X \in \mathbb{R}^{B \times D}$

vectorize

vectorize

vectorize



$$\mathbf{m}[j] = \frac{1}{B}\sum_{i=1}^{B} X[i,j]$$

$$\mathbf{v}[j] = \frac{1}{B}\sum_{i=1}^{B} (X[i,j] - \mathbf{m}[j])^2$$

normalized activations:

$$\widehat{X}[i,j] = \boldsymbol{\gamma}[j]\frac{X[i,j] - \mathbf{m}[j]}{\sqrt{\mathbf{v}[j] + \epsilon}} + \boldsymbol{\beta}[j]$$

Batch Normalization

Suppose we have a batch of activations $X \in \mathbb{R}^{B \times D}$

vectorize

vectorize

vectorize

$$\mathbf{m}[j] = \frac{1}{B}\sum_{i=1}^{B} X[i,j]$$

$$\mathbf{v}[j] = \frac{1}{B}\sum_{i=1}^{B} (X[i,j] - \mathbf{m}[j])^2$$

normalized activations:

$$\widehat{X}[i,j] = \boldsymbol{\gamma}[j]\frac{X[i,j] - \mathbf{m}[j]}{\sqrt{\mathbf{v}[j] + \epsilon}} + \boldsymbol{\beta}[j]$$

Learnable Parameters

Batch Normalization

Each column is a feature. Each row is an example.

$$X \in \mathbb{R}^{B \times D}$$

$$\mathbf{m}[j] = \frac{1}{B}\sum_{i=1}^{B} X[i,j], \qquad \mathbf{v}[j] = \frac{1}{B}\sum_{i=1}^{B}(X[i,j] - \mathbf{m}[j])^2$$

$$\widehat{X}[i,j] = \boldsymbol{\gamma}[j]\frac{X[i,j] - \mathbf{m}[j]}{\sqrt{\mathbf{v}[j] + \epsilon}} + \boldsymbol{\beta}[j]$$

Learnable scale γ and shift β act on the normalized feature.

Batch Normalization

Training

Use the current mini-batch mean and variance. Update running estimates.

$$\mathbf{m}_{\text{run}} \leftarrow (1-\alpha)\mathbf{m}_{\text{run}} + \alpha\mathbf{m}_{\text{batch}}$$

$$\mathbf{v}_{\text{run}} \leftarrow (1-\alpha)\mathbf{v}_{\text{run}} + \alpha\mathbf{v}_{\text{batch}}^{\text{unbiased}}$$

Evaluation

Use the saved running statistics. Keep the learned γ and β.

The running-variance update above follows the PyTorch convention.

Batch Normalization

Training and evaluation behave differently

model.train(): use batch statistics and update running statistics.

model.eval(): use saved running statistics.

Small or unrepresentative batches can make BatchNorm unreliable.

In CNNs, compute one set of statistics per channel across batch and space.

LayerNorm and RMSNorm use features within each example or token.

They are common alternatives in Transformers. We will study them later.

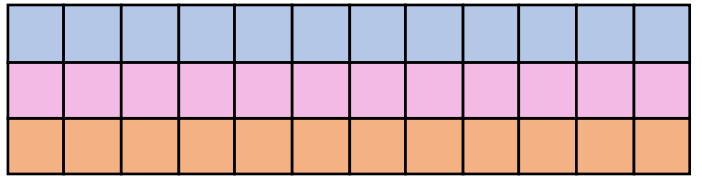
PyTorch defaults: track_running_stats=True.

Outline

- Multilayer Perceptron (MLP)
  - Linear Layer
  - Activation Functions
  - Batch Normalization
  - **Dropout**
  - Build a Deep MLP

Dropout

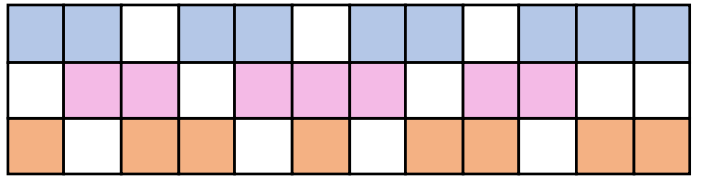
Suppose we have a batch of activations $X \in \mathbb{R}^{B \times D}$



Dropout

Suppose we have a batch of activations $X \in \mathbb{R}^{B \times D}$

We can make them stochastic by randomly turning some units off

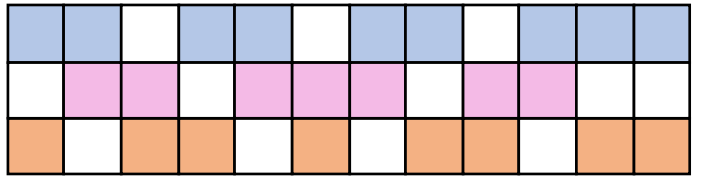


Dropout

Suppose we have a batch of activations $X \in \mathbb{R}^{B \times D}$

We can make them stochastic by randomly turning some units off

Mathematically, we create a matrix mask $M \in \mathbb{R}^{B \times D}$ as follows



$$M[i,j] \sim \text{Bernoulli}(1-p)$$

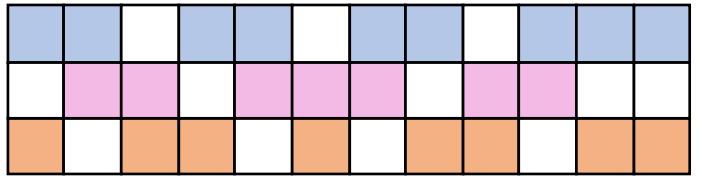
$$\mathbb{P}(M[i,j] = 1) = 1 - p$$
$$\mathbb{P}(M[i,j] = 0) = p$$

Dropout

Suppose we have a batch of activations $X \in \mathbb{R}^{B \times D}$

We can make them stochastic by randomly turning some units off

Mathematically, we create a matrix mask $M \in \mathbb{R}^{B \times D}$ as follows



$$M[i,j] \sim \text{Bernoulli}(1-p)$$

$$\mathbb{P}(M[i,j]=1) = 1-p$$
$$\mathbb{P}(M[i,j]=0) = p$$

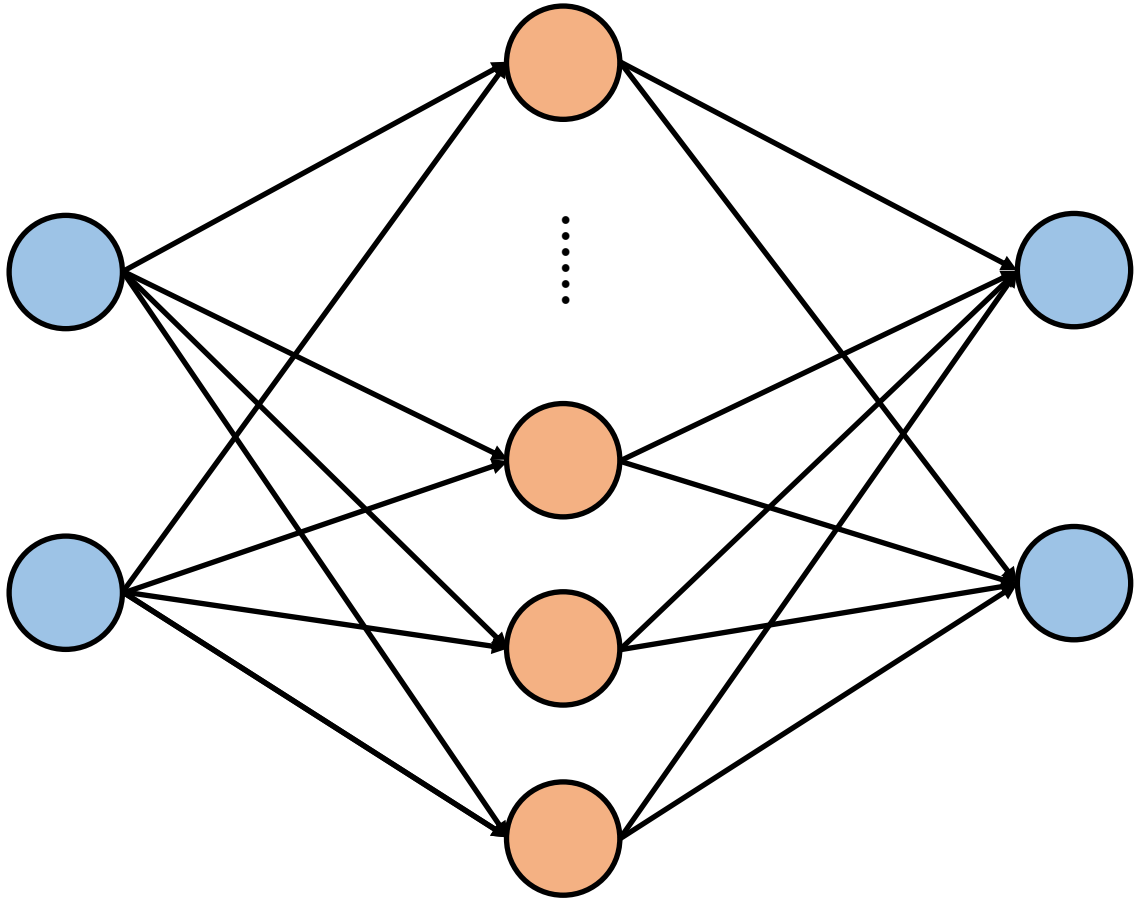
Then we perform element-wise product (a.k.a. Hadamard product)

$$\hat{X} = M \odot X$$

$$\hat{X}[i,j] = M[i,j]X[i,j]$$

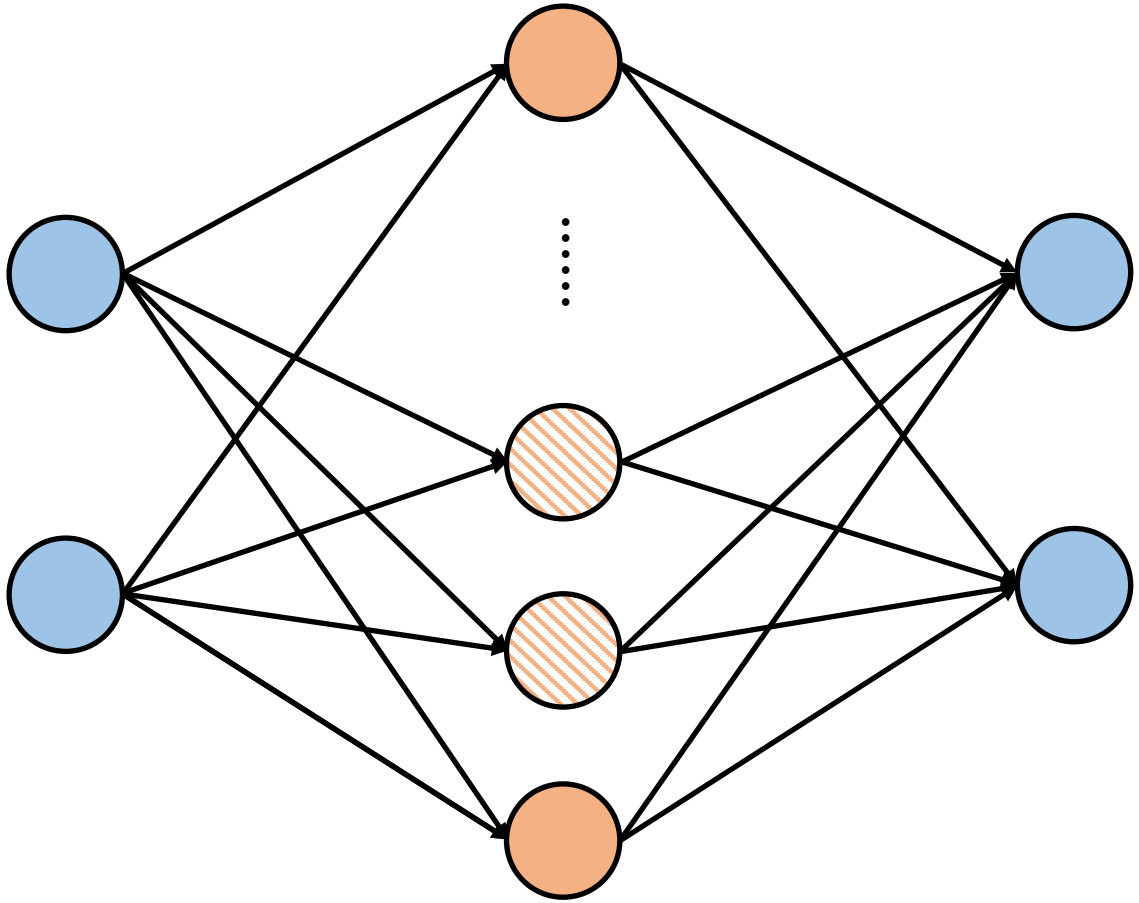
Dropout

What does Dropout [8] do to the neural networks?



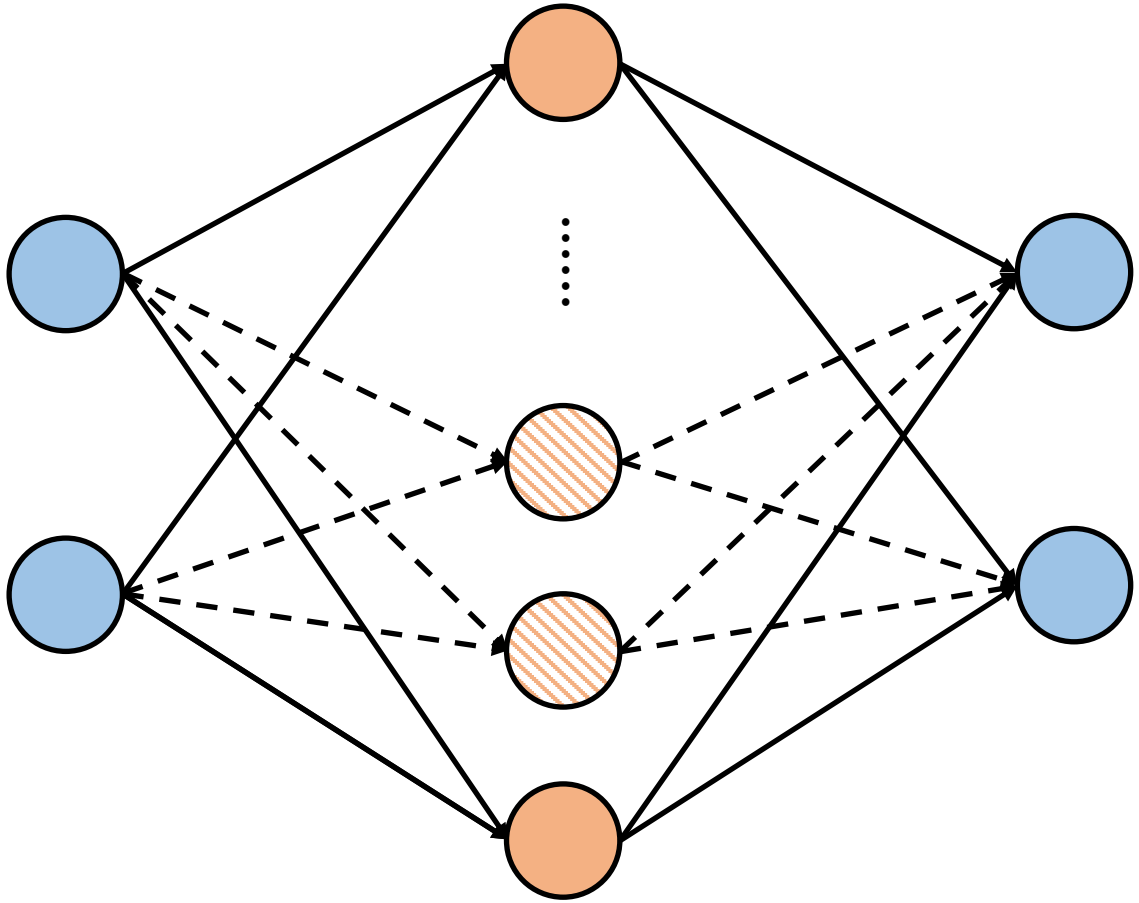
Dropout

What does Dropout [8] do to the neural networks?



Dropout

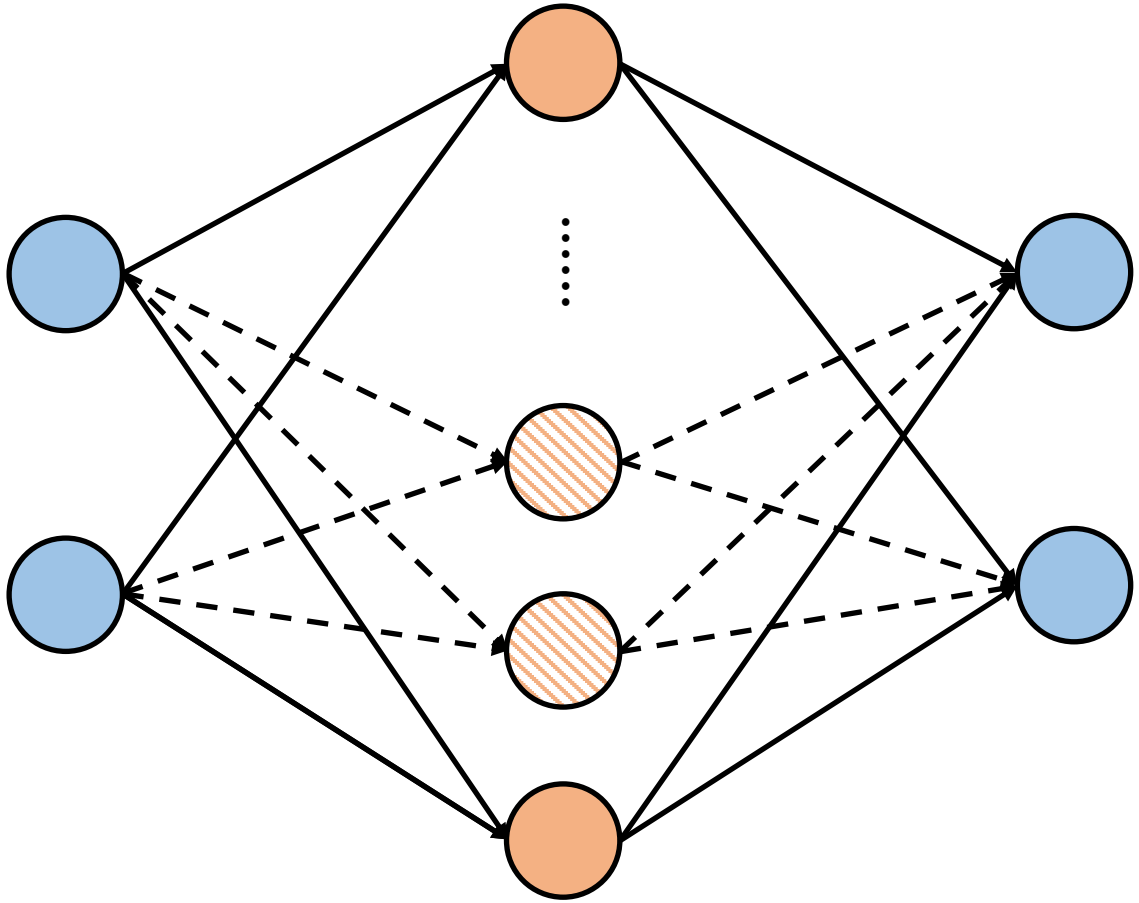
What does Dropout [8] do to the neural networks?



Dropout

What does Dropout [8] do to the neural networks?

It turns off a random subset of units, thus blocking a random subset of paths!

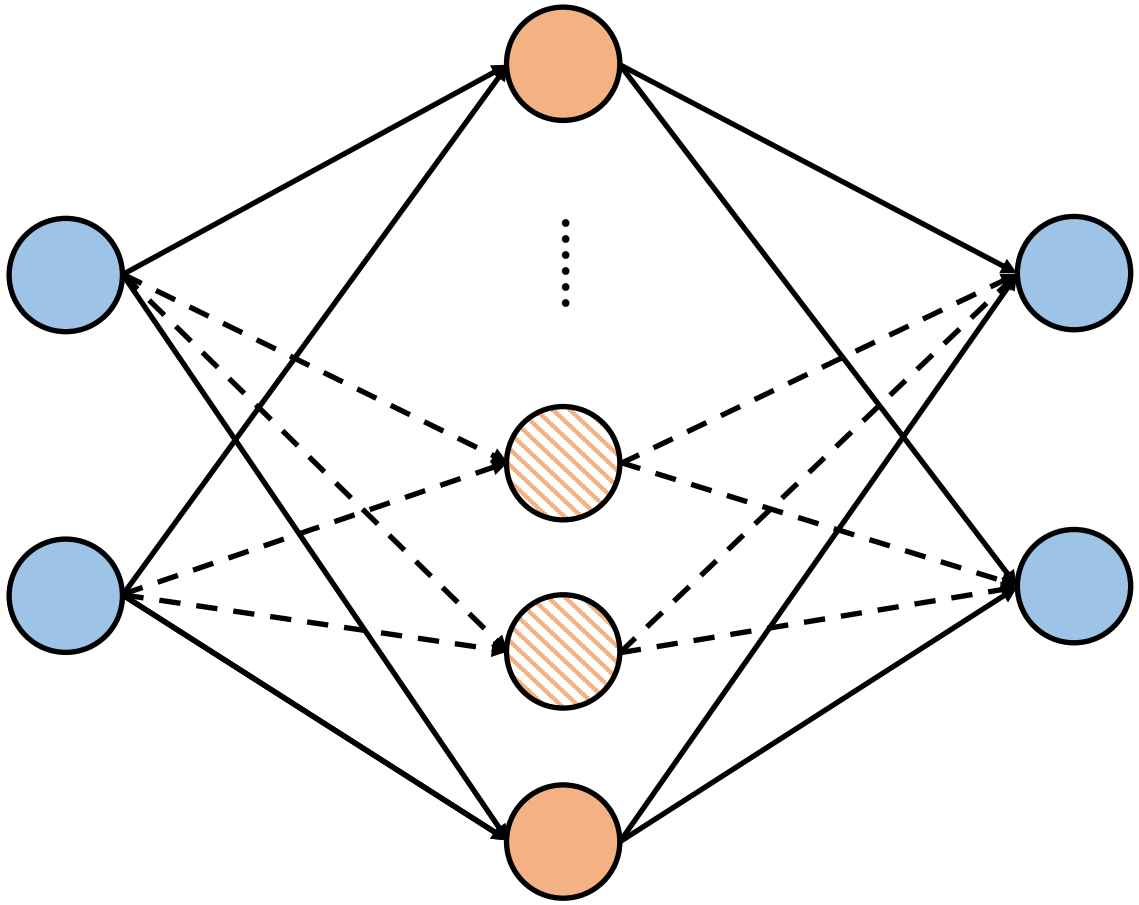


Dropout

What does Dropout [8] do to the neural networks?

It turns off a random subset of units, thus blocking a random subset of paths!

This discourages reliance on particular units and can reduce overfitting.



Dropout

How do training and evaluation use dropout?

Dropout

How do training and evaluation use dropout?

The original convention (p = drop probability)

Match the mean of each masked activation at evaluation time.

$$M_{ij} \sim \text{Bernoulli}(1-p), \qquad \widetilde{X}_{ij} = M_{ij} X_{ij}$$

Dropout

How do training and evaluation use dropout?

The original convention (p = drop probability)

Match the mean of each masked activation at evaluation time.

$$M_{ij} \sim \text{Bernoulli}(1-p), \qquad \widetilde{X}_{ij} = M_{ij} X_{ij}$$

$$\mathbb{E}_M[\widetilde{X}_{ij} \mid X] = (1-p) X_{ij}$$

Dropout

How do training and evaluation use dropout?

The original convention (p = drop probability)

Match the mean of each masked activation at evaluation time.

$$M_{ij} \sim \text{Bernoulli}(1-p), \qquad \widetilde{X}_{ij} = M_{ij} X_{ij}$$

$$\mathbb{E}_M[\widetilde{X}_{ij} \mid X] = (1-p) X_{ij}$$

At evaluation, use (1 − p)X instead of sampling masks.

This matches a local mean, not the exact average of a nonlinear network.

Dropout

Inverted dropout

Scale the surviving activations during training.

$$q = 1 - p > 0, \qquad M_{ij} \sim \text{Bernoulli}(q), \qquad \widetilde{X} = \frac{M \odot X}{q}$$

Dropout

Inverted dropout

Scale the surviving activations during training.

$$q = 1 - p > 0, \qquad M_{ij} \sim \text{Bernoulli}(q), \qquad \widetilde{X} = \frac{M \odot X}{q}$$

$$\mathbb{E}_M[\widetilde{X} \mid X] = X$$

Evaluation uses X directly. No mask and no extra scaling.

This is the convention used by torch.nn.Dropout.

Outline

- Multilayer Perceptron (MLP)
  - Linear Layer
  - Activation Functions
  - Batch Normalization
  - Dropout
  - **Build a Deep MLP**

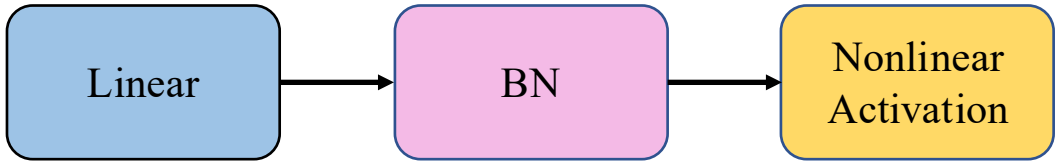
Build a Deep MLP

We can combine linear layers, activations, normalization, and dropout into a deep MLP.

Build a Deep MLP

We can combine linear layers, activations, normalization, and dropout into a deep MLP.

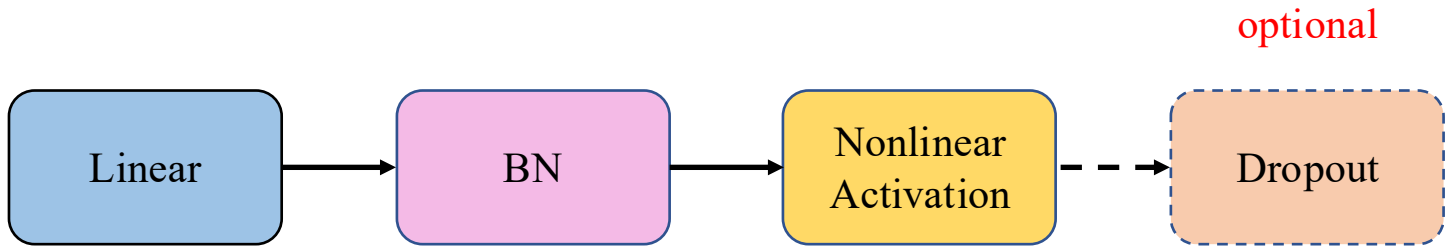
One possible hidden block (normalization and dropout are optional):



Build a Deep MLP

We can combine linear layers, activations, normalization, and dropout into a deep MLP.

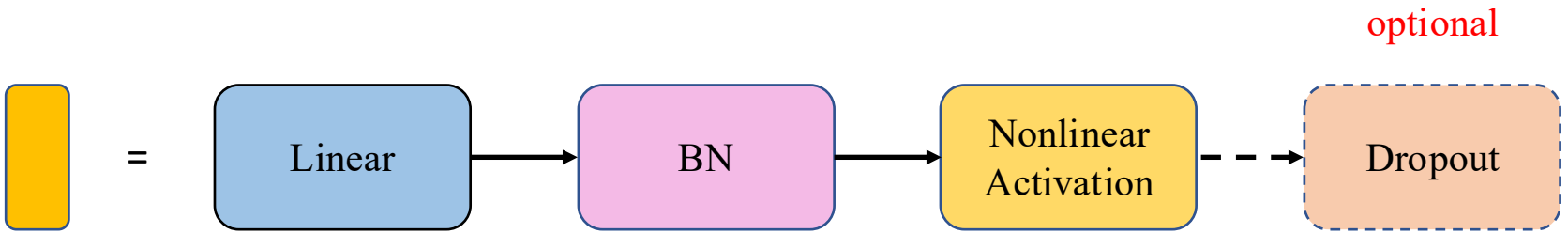
One possible hidden block (normalization and dropout are optional):



Build a Deep MLP

We can combine linear layers, activations, normalization, and dropout into a deep MLP.

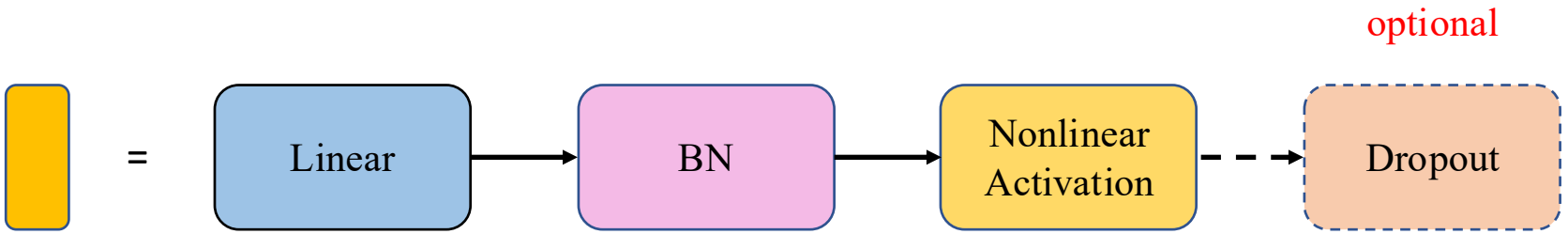
One possible hidden block (normalization and dropout are optional):



Build a Deep MLP

We can combine linear layers, activations, normalization, and dropout into a deep MLP.

One possible hidden block (normalization and dropout are optional):



Stack hidden blocks, then choose an output layer for the task:

References

[1] Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. ICML.

[2] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. ICCV.

[3] Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (GELUs). arXiv:1606.08415.

[4] Ramachandran, P., Zoph, B., & Le, Q. V. (2017). Searching for activation functions. arXiv:1710.05941.

[5] Shazeer, N. (2020). GLU variants improve Transformer. arXiv:2002.05202.

References

[6] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. ICML.

[7] Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? NeurIPS.

[8] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. JMLR, 15, 1929–1958.

[9] Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). Maxout networks. ICML.

[10] Shen, Y., Tan, S., Sordoni, A., & Courville, A. (2019). Ordered neurons: Integrating tree structures into recurrent neural networks. ICLR.

Questions?