CS769 Advanced NLP

Deep Learning Basic

Junjie Hu

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Slides adapted from Graham
https://junjiehu.github.io/cs769-fall23/

Logistics

- HW1 will be due at 11:59PM on Sept 20. Details can be found here: https://github.com/JunjieHu/cs769-assignments/tree/main/assignment1

- HW1 will be submitted to Canvas.

- Individual assignments (HW1, 2, 4) will have 3 late days in total, and group assignments (HW3, 5) will have another 3 late days in total.

- Bonus point (up to 5%) if you actively answer questions on Piazza (endorsed by this instructor more than 5 times).

Goals for Today

- Tensors and numerical computation
- **Model**: Define a **neural network architecture**
- **Forward**: computing predictions & Loss
- **Backpropagation**: computing gradients
- **Optimization**: parameter update
- Training Tricks

Neural Network Frameworks

theano

∂y/net

Caffe

Chainer

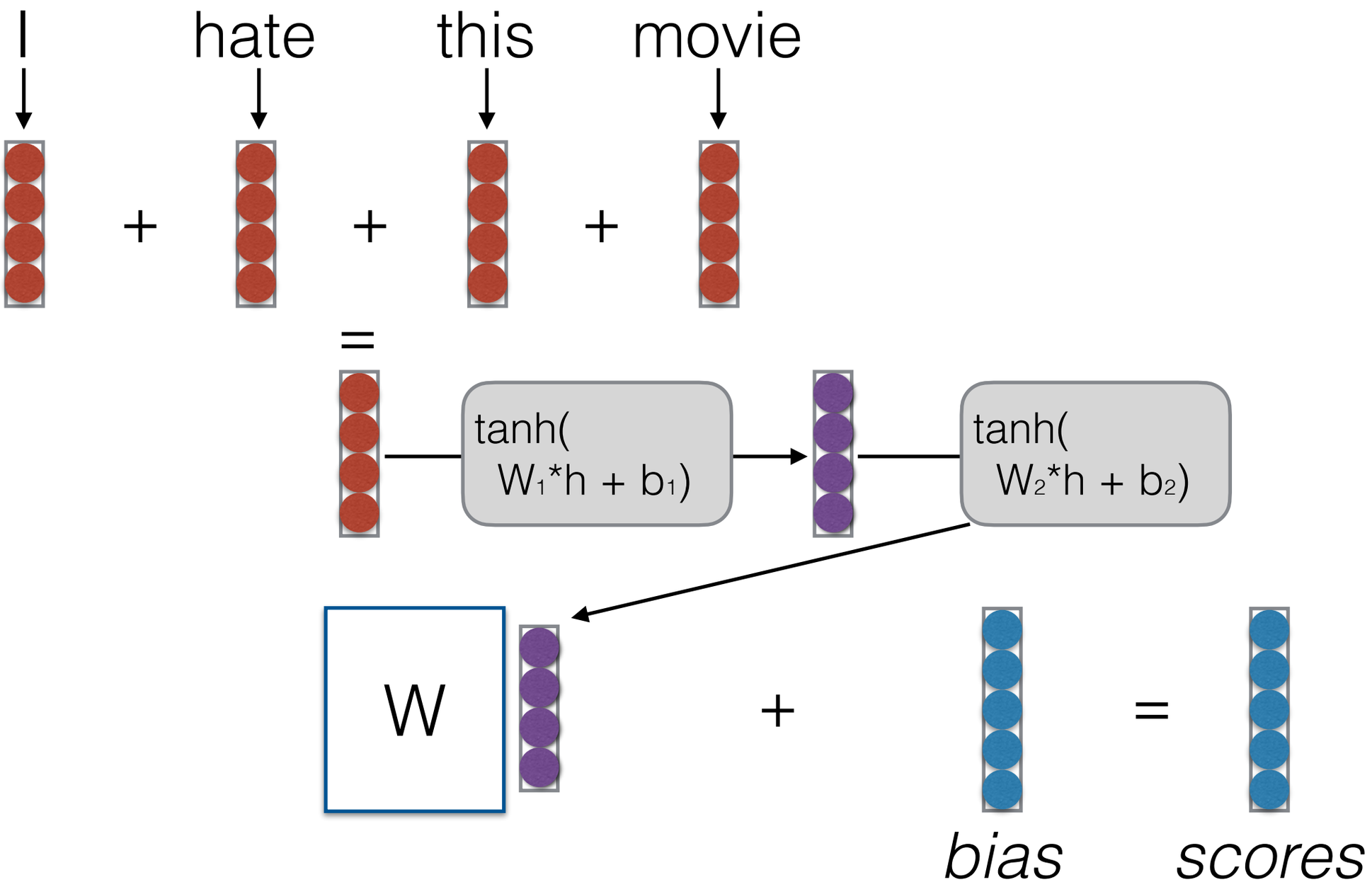
mxnet

PYTORCH

TensorFlow

JAX

Example (HW1): Deep CBOW Model



Mikolov, et al. CBOW: Efficient Estimation of Word Representations in Vector Space

NN App Algorithm Sketch

- Create a model and define a loss (i.e., **construct a computation graph**)
- For each example
  - **Forward: calculate the result** (prediction & loss) of that computation
  - if training
    - perform **back propagation**
    - **update** parameters

Tensors and Numerical Computation

Numerical Computation Backend

- Most neural network libraries use a backend for numerical computation
- **PyTorch/Tensorflow:** MKL, CUDNN, CUDA, OpenMP, custom-written kernels
- Support many numerical functions on **tensors**

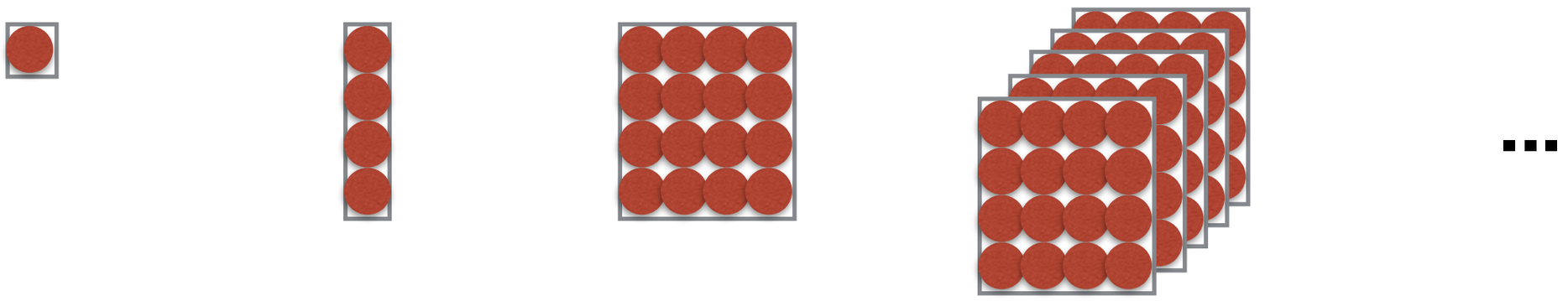
```
import torch
import torch.nn as nn
print(*torch.__config__.show().split("\n"), sep="\n")
```

```
PyTorch built with:
  - GCC 7.3
  - C++ Version: 201402
  - Intel(R) Math Kernel Library Version 2020.0.0 Pro
  - Intel(R) MKL-DNN v2.2.3 (Git Hash 7336ca9f055cf1b
  - OpenMP 201511 (a.k.a. OpenMP 4.5)
  - LAPACK is enabled (usually provided by MKL)
  - NNPACK is enabled
  - CPU capability usage: AVX2
  - CUDA Runtime 11.1
  - NVCC architecture flags: -gencode;arch=compute_37
  - CuDNN 8.0.5
  - Magma 2.5.2
  - Build settings: BLAS_INFO=mkl, BUILD_TYPE=Release
```

Tensors

- An n-dimensional array

Scalar **Vector** **Matrix** **3-dim Tensor**



...

$x \in \mathbb{R}$ $\mathbf{x} \in \mathbb{R}^4$ $\mathbf{X} \in \mathbb{R}^{4\times4}$ $\mathbf{X} \in \mathbb{R}^{4\times4\times5}$

- Widely used in neural networks

- Parameters in NNs consist of different shape of tensors, which store both their **values** and **gradients**

Tensor Operations

- **PyTorch/Tensorflow:** Support many different matrix operations: matrix-multiply

```
import numpy as np
x = torch.Tensor([[2, 3], [1, 2]])
x = torch.Tensor(np.array([[-1, 1], [2, 4]]))
x = torch.zeros([2, 3], dtype=torch.int32)
```

create tensors from list, numpy.array

```
import torch
import torch.nn as nn

x = torch.randn((4, 2))
W = torch.randn((3, 4))
print(x)
print(W)
```

```
tensor([[-1.5372,  0.0845],
        [ 0.5752,  0.7634],
        [ 0.4265, -0.1287],
        [-1.8629, -0.8520]])
tensor([[-1.1272, -1.1810,  0.0867,  0.0676],
        [-0.1070,  3.2586,  0.7446, -1.2094],
        [-1.7670,  0.2900, -1.0881, -1.5555]])
```

```
[10] torch.matmul(W, x)  # results in a [3,2] matrix
```

```
tensor([[ 0.9646, -1.0656],
        [ 4.6092,  3.4132],
        [ 5.3167,  1.5373]])
```

matrix multiply

```
x = torch.randn((4,2))
z = torch.randn((4,2))
print(x)
print(z)
```

```
tensor([[ 0.0762,  1.5145],
        [-0.4747, -0.9141],
        [ 0.7106,  0.4888],
        [ 0.6959, -0.5305]])
tensor([[-0.1766,  0.6187],
        [ 0.9254, -0.5931],
        [-0.9162,  0.3209],
        [ 0.0216, -0.7116]])
```

```
[13] x * z   # results in a [4,2] matrix
```

```
tensor([[-0.0135,  0.9371],
        [-0.4392,  0.5421],
        [-0.6510,  0.1569],
        [ 0.0151,  0.3775]])
```

Element-wise matrix multiply

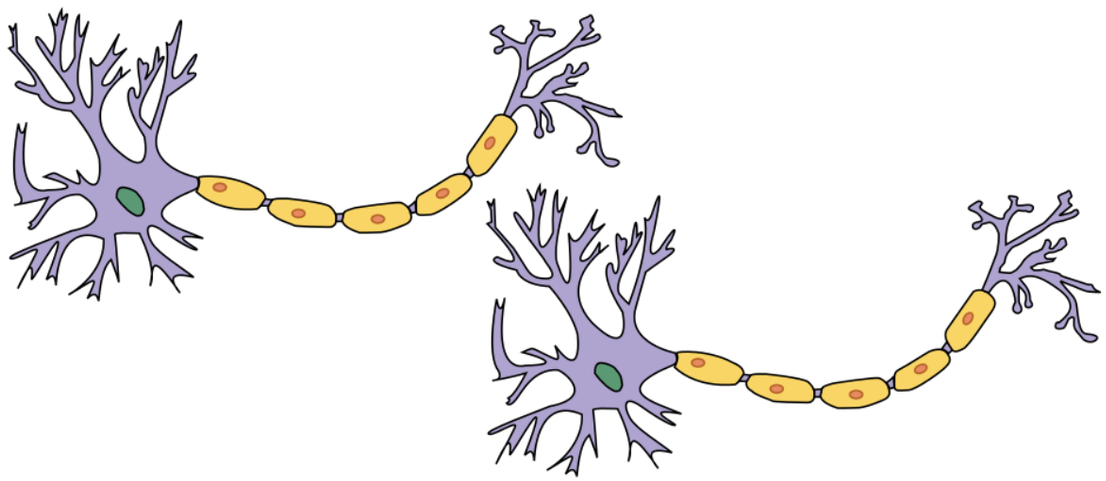
Model and Parameter Definition

NN App Algorithm Sketch

- Create a model and define a loss (i.e., **construct a computation graph**)
- For each example
  - **Forward: calculate the result** (prediction & loss) of that computation
  - if training
    - perform **back propagation**
    - **update** parameters

“Neural” Nets

Original Motivation: Neurons in the Brain



Current Conception: Computation Graphs

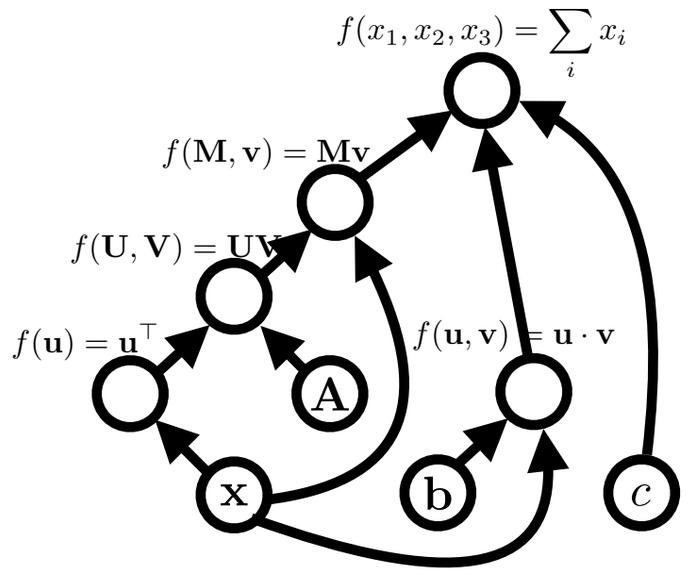
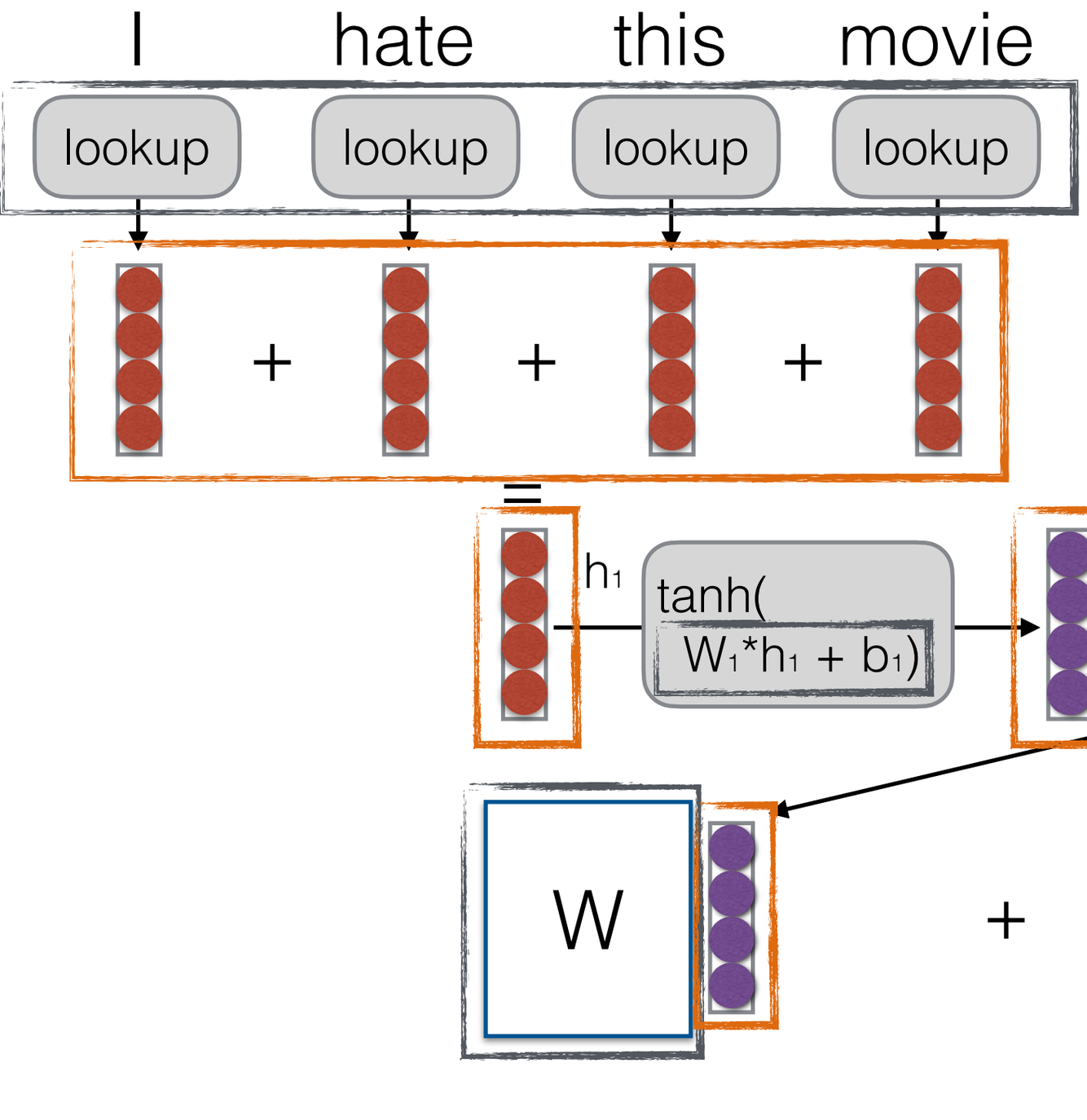


Image credit: Wikipedia

Tensors in Neural Networks



Can represent

parameters

intermediate values

Example Model Creation

- Define model's parameters:
  - Weight matrix W_0 for the input embedding layer
  - Weight matrix W_1, W_2 and bias b_1, b_2 for feedforward layers
  - Any other layers …

```
class DanModel(BaseModel):
    def __init__(self, args, vocab, tag_size):
        super(DanModel, self).__init__(args, vocab, tag_size)
        self.define_model_parameters()
        self.init_model_parameters()

        # Use pre-trained word embeddings if emb_file exists
        if args.emb_file is not None:
            self.copy_embedding_from_numpy()
```

```
    def define_model_parameters():
        """
        Define the model's parameters, e.g., embedding layer,
        """
        raise NotImplementedError()
```

Parameter Initialization

- Neural nets must have weights that are not identical to learn non-identical features

- **Uniform Initialization:** Initialize weights in some range, such as $[-v, v], v = 0.01$ for example
    - *Problem!* Depending on the size of the net, inputs to downstream nodes may be very large

- **Glorot (Xavier) Initialization, He Initialization:** Initialize based on the size of the matrix

Glorot Init: $v = \sqrt{\frac{6}{d_{\text{in}} + d_{\text{out}}}}$

Example Model Initialization

- Initialize model's parameters using **Glorot** or others:
  - Create initial random values within [-v, v] for tensors such as W_0, W_1, W_2, b_1, b_2 , and any others before training.

```
def init_model_parameters(self):
    """
    Initialize the model's parameters by uniform sampling from a range [-v, v]
    """
    raise NotImplementedError()
```

Loss Function

- Given a labeled example (x, y^*), we use a NN $f(x)$ to estimate the condition probability and predict the label as $y = \arg\max P_\theta(y|x)$. Then we compute how close our predict w.r.t. the true label by a loss function.

$$P_\theta(y|x) = \text{Softmax}(f(x))$$

- **Classification**: Cross-Entropy

$$\mathcal{L}(x, y^*) = -\log P_\theta(y = y^*|x)$$

- **Regression**: L1 loss, L2 loss (a.k.a. Mean Square Error)

$$\mathcal{L}(x, y^*) = \|f(x) - y^*\|_1$$
$$\mathcal{L}(x, y^*) = \|f(x) - y^*\|_2$$

Forward Propagation:
Computing Predictions & Loss

NN App Algorithm Sketch

- Create a model and define a loss (i.e., **construct a computation graph**)
- For each example
- **Forward: calculate the result** (prediction & loss) of that computation
- if training
  - perform **back propagation**
  - **update** parameters

expression:

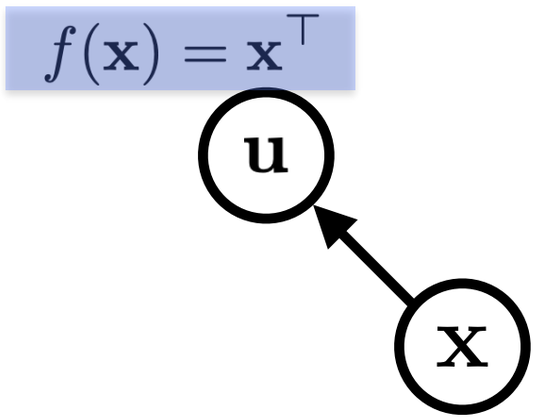
$$\mathbf{x}$$

graph:

A **node** is a {tensor, matrix, vector, scalar} value

An **edge** represents a function argument (and also a data dependency). They are just pointers to nodes.

A **node** with an incoming **edge** is a **function** of that edge's tail node.

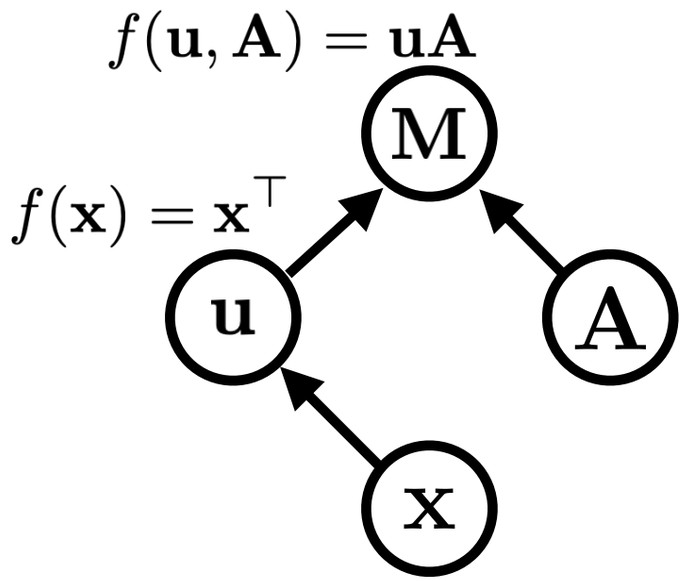


expression:

$$\mathbf{x}^\top \mathbf{A}$$

graph:

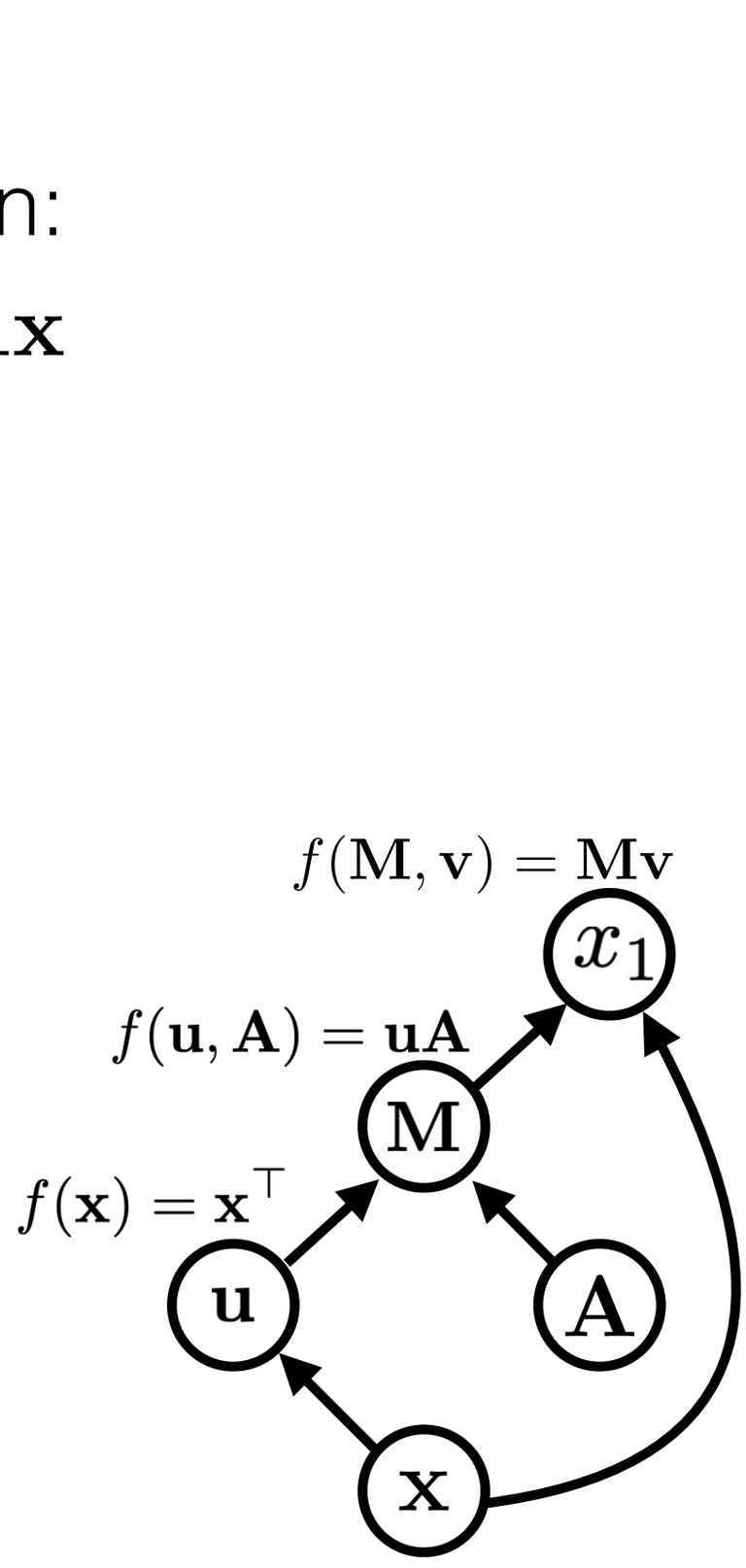
Functions can be unary, binary, … *n*-ary (no. of inputs).
Often they are unary or binary.



expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x}$$

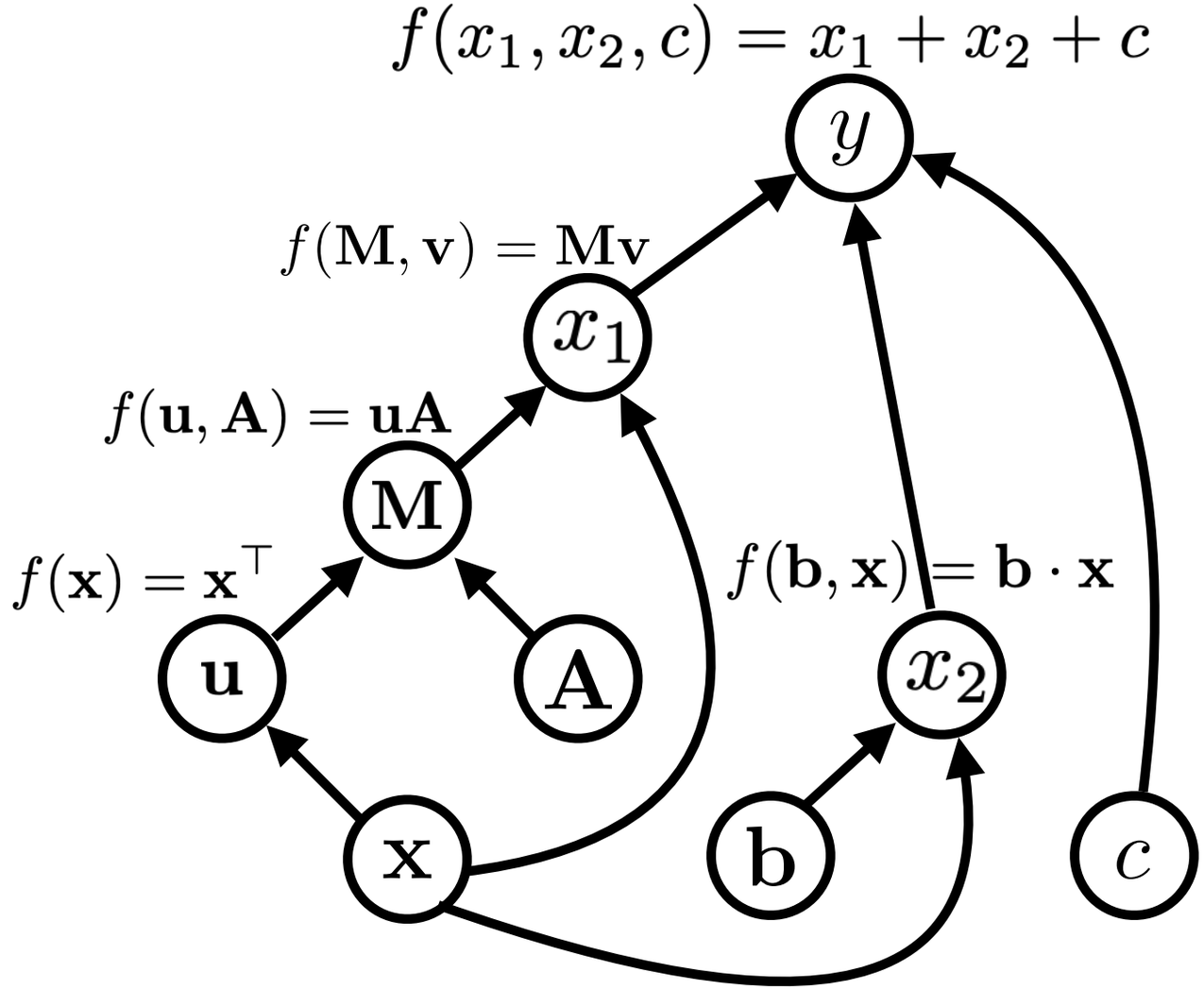
graph:



expression:

$$\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

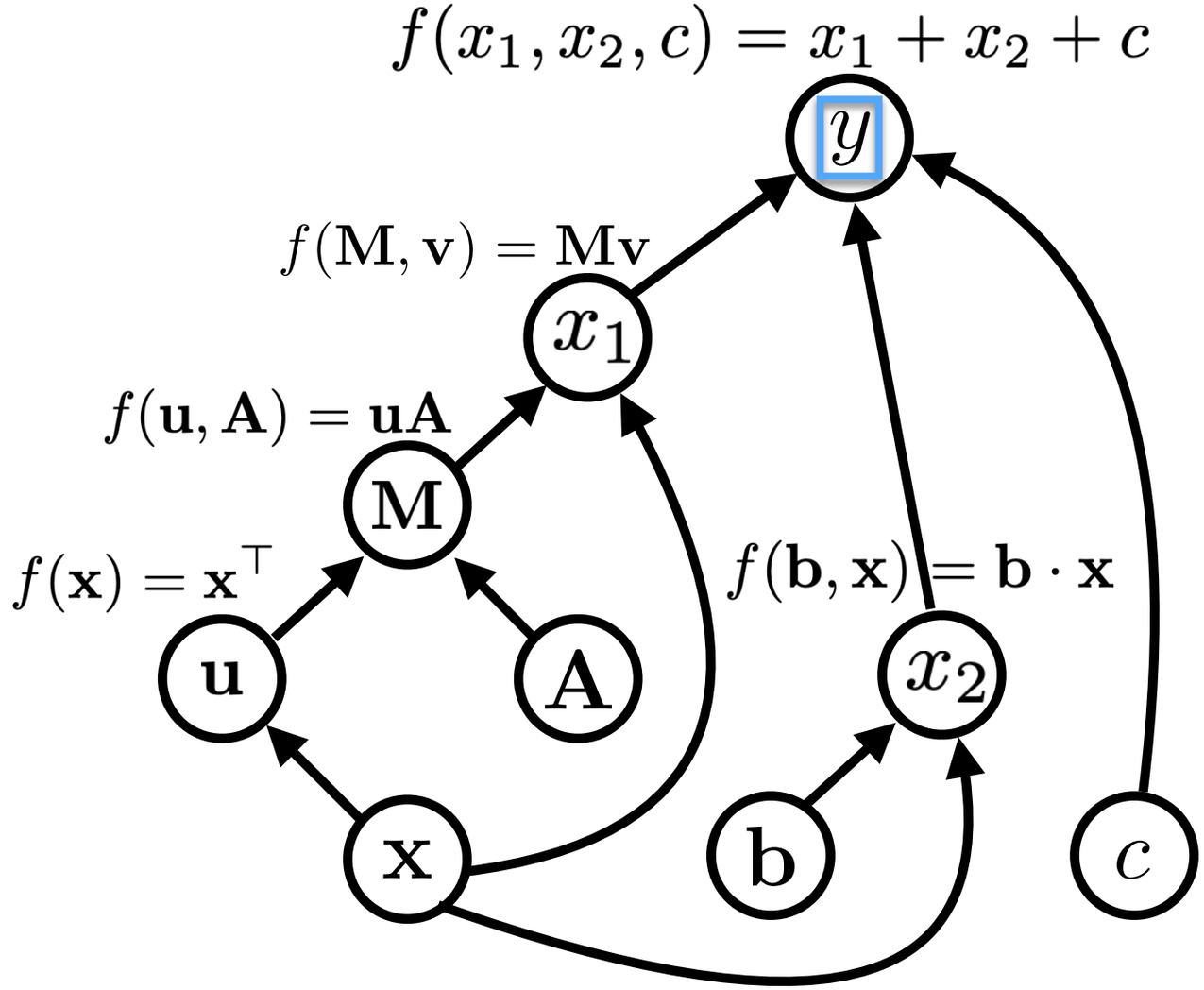
graph:



expression:

$$y = \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

graph:

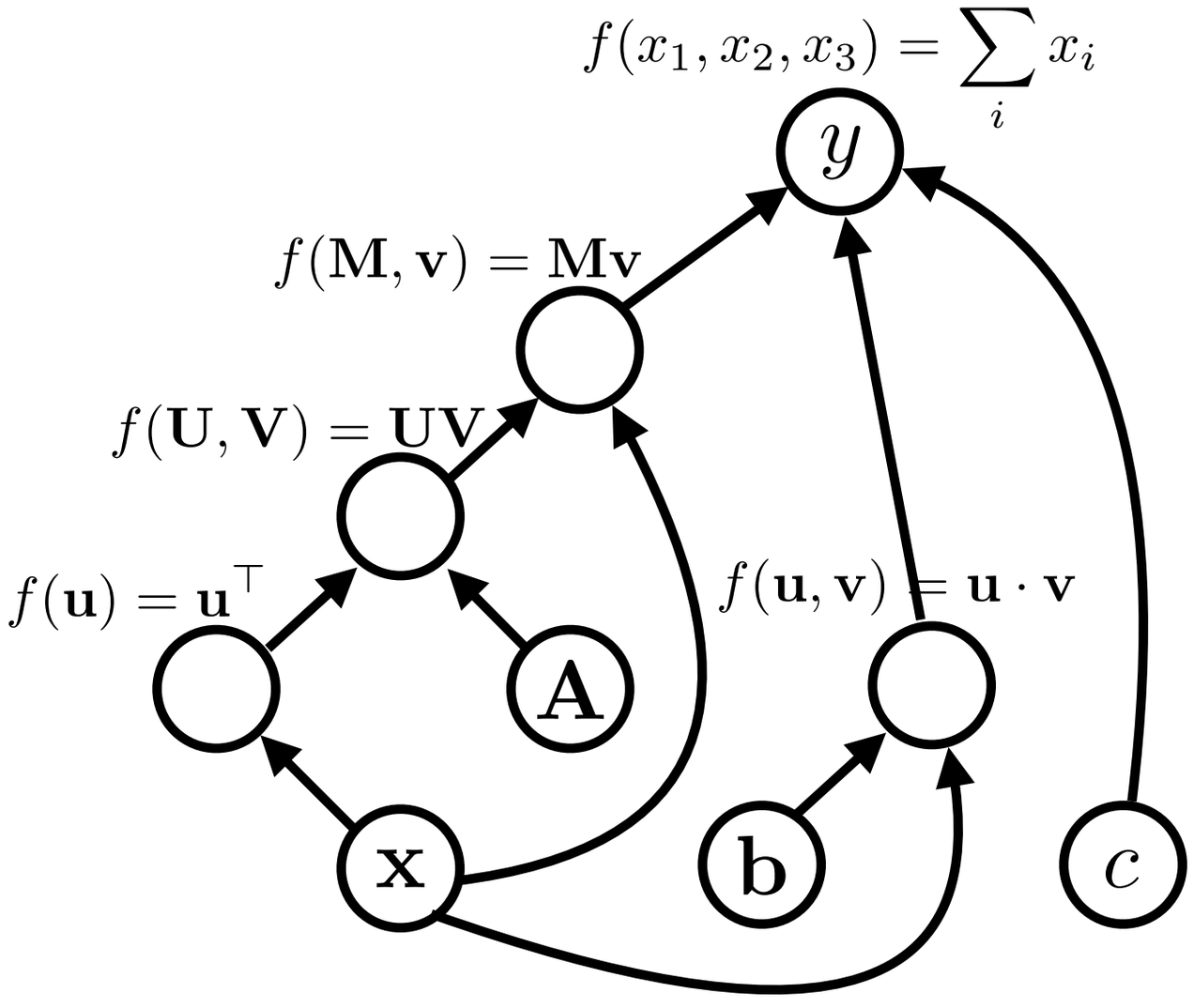


Computation graphs are generally directed and acyclic

expression:

$$y = \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

graph:

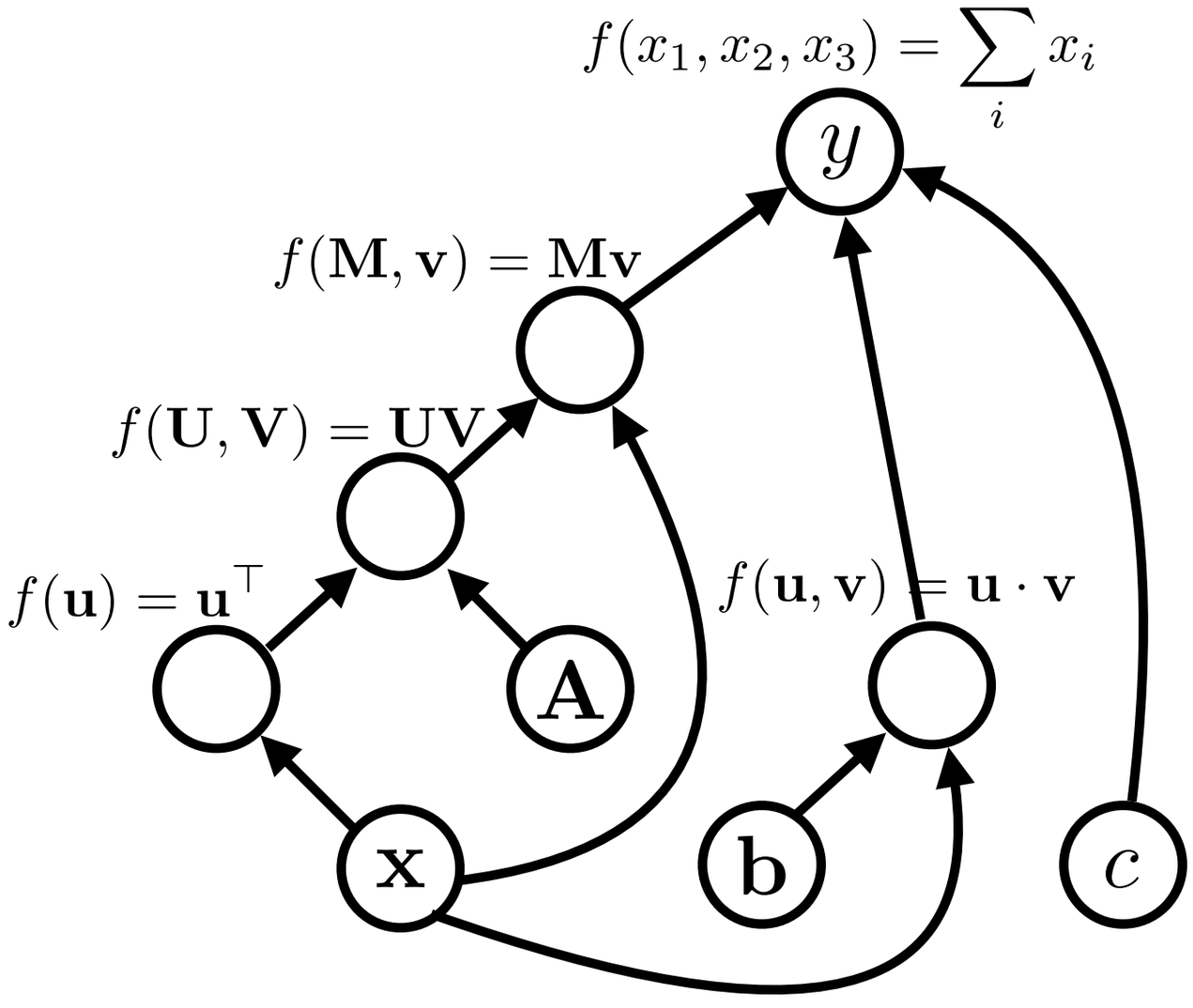



```python
# x: 5-dimensional vector
# Model's parameters contain:
#    A: 5x5 matrix
#    b: 5-dimensional vector
#    c: scalar

class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.define_model_parameters()

    def define_model_parameters(self):
        self.A = nn.Parameter(torch.randn((5,5)))
        self.b = nn.Parameter(torch.randn((5)))
        self.c = nn.Parameter(torch.randn((1)))

    def forward(self, x):
        U = x.T
        print('U', U)
        M = torch.matmul(U, self.A)
        print('M', M)
        Mv = torch.matmul(M, x)
        print('Mv', Mv)
        bx = torch.dot(self.b, x)
        print('bx', bx)
        y = Mv + bx + self.c
        return y

x = torch.randn((5))
model = SimpleModel()
y = model(x)
print(y)
```

expression:

$$y = \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

graph:




```
# x: 5-dimensional vector
# Model's parameters contain:
#    A: 5x5 matrix
#    b: 5-dimensional vector
#    c: scalar

class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.define_model_parameters()

    def define_model_parameters(self):
        self.A = nn.Parameter(torch.randn((5,5)))
        self.b = nn.Parameter(torch.randn((5)))
        self.c = nn.Parameter(torch.randn((1)))

    def forward(self, x):
        U = x.T
        print('U', U)
        M = torch.matmul(U, self.A)
        print('M', M)
        Mv = torch.matmul(M, x)
        print('Mv', Mv)
        bx = torch.dot(self.b, x)
        print('bx', bx)
        y = Mv + bx + self.c
        return y

x = torch.randn((5))
model = SimpleModel()
y = model(x)
print(y)
```

Colab: https://colab.research.google.com/drive/12LyIcg90fm6ed2W1hfkOMGebBkVhopaN?usp=sharing

Operations

- Operations must know:
- **Forward:** Calculate their predictions/loss given input
  - Prediction: $y = f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$
  - Loss: $\mathcal{L} = (y - y^*)^2$
- **Backward:** Calculate the derivative of model parameters w.r.t. the loss

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}}, \quad \frac{\partial \mathcal{L}}{\partial c}$$

Back Propagation

NN App Algorithm Sketch

- Create a model and define a loss (i.e., **construct a computation graph**)
- For each example
  - **Forward: calculate the result** (prediction & loss) of that computation
  - if training
    - perform **back propagation**
    - **update** parameters

Back Propagation

- Mean-square error: $\mathcal{L} = (y - y^*)^2$

graph:

$$y = f(x_1, x_2, c) = x_1 + x_2 + c$$

$$\frac{\partial \mathcal{L}}{\partial y} = 2(y - y^*)$$

$$\frac{\partial \mathcal{L}}{\partial x_1} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial x_1} = 2(y - y^*)$$

$$f(\mathbf{M}, \mathbf{x}) = \mathbf{M}\mathbf{x}$$

$$f(\mathbf{u}, \mathbf{A}) = \mathbf{u}\mathbf{A}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{M}} = \frac{\partial \mathcal{L}}{\partial x_1} \cdot \frac{\partial x_1}{\partial \mathbf{M}} = 2(y - y^*)\mathbf{x}$$

$$f(\mathbf{x}) = \mathbf{x}^\top$$

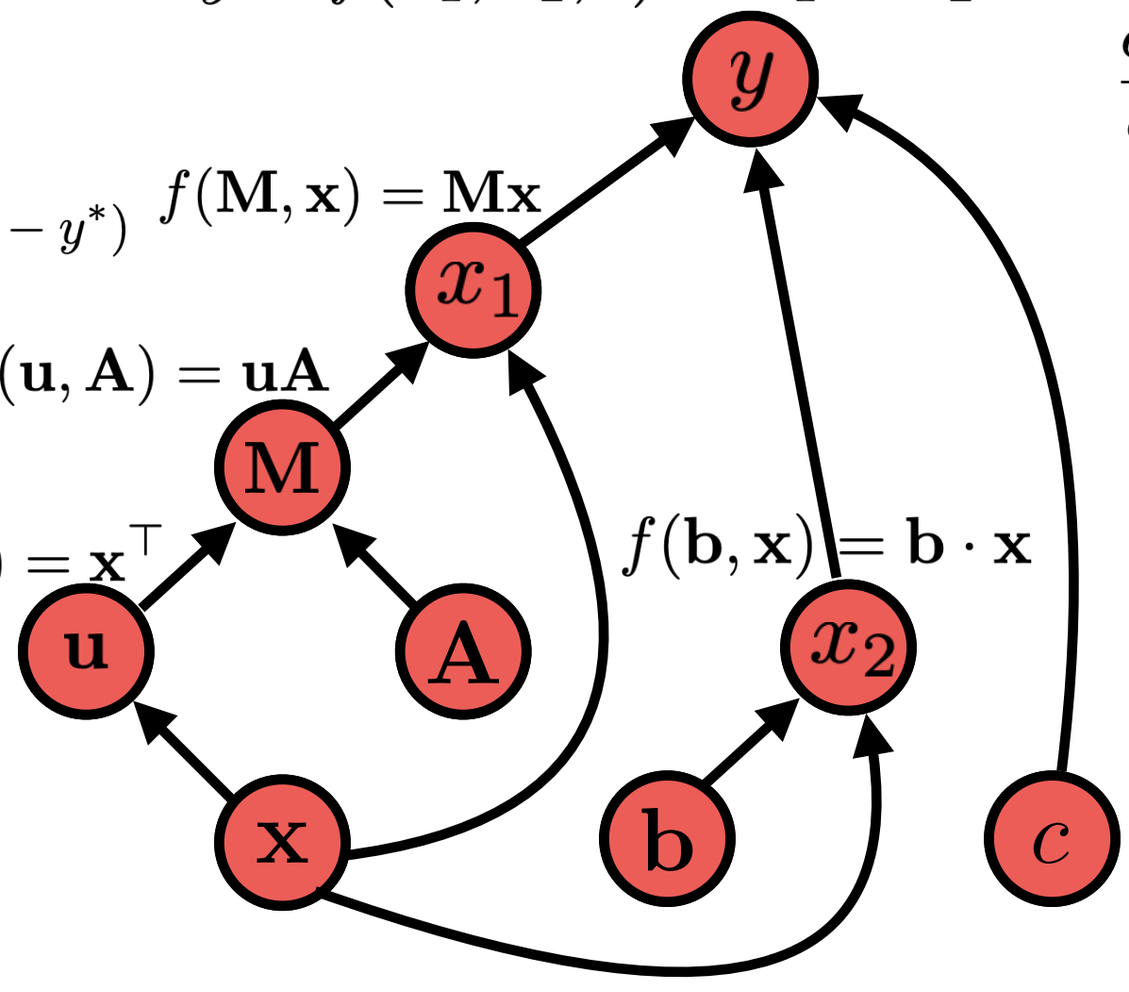
$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \frac{\partial \mathcal{L}}{\partial \mathbf{M}} \cdot \frac{\partial \mathbf{M}}{\partial \mathbf{A}} = 2(y - y^*)\mathbf{x}\mathbf{x}^\top$$

$$f(\mathbf{b}, \mathbf{x}) = \mathbf{b} \cdot \mathbf{x}$$

$$\frac{\partial \mathcal{L}}{\partial x_2} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial x_2} = 2(y - y^*)$$

$$\frac{\partial \mathcal{L}}{\partial c} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial c} = 2(y - y^*)$$

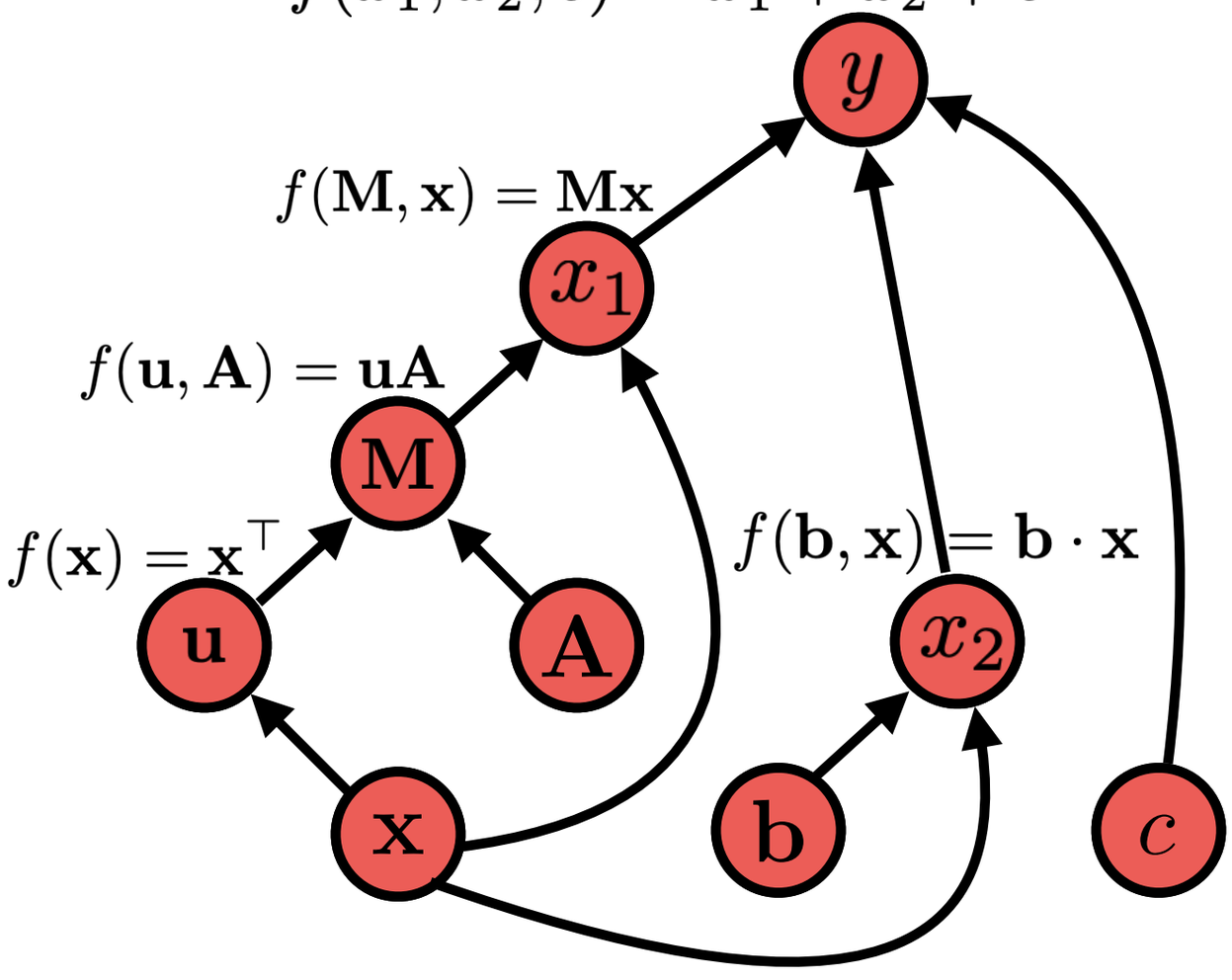
$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}}{\partial x_2} \cdot \frac{\partial x_2}{\partial \mathbf{b}} = 2(y - y^*)\mathbf{x}$$



Model parameters: $\mathbf{A}, \mathbf{b}, c$

Back Propagation

graph:



- Mean-square error: $\mathcal{L} = (y - y^*)^2$

```
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.01)
loss_func = nn.MSELoss()

y_true = torch.FloatTensor([1.0]) # ground true
loss = loss_func(y, y_true)
print(loss)
```

```
tensor(34.0528, grad_fn=<MseLossBackward0>)
```

- Compute gradients automatically by most tools: $\frac{\partial \mathcal{L}}{\partial A}, \frac{\partial \mathcal{L}}{\partial b}, \frac{\partial \mathcal{L}}{\partial c}$

```
optimizer.zero_grad()
loss.backward(retain_graph=True)
for name, param in model.named_parameters():
  print(name, param.grad)
```

```
A tensor([[ -0.0313,  -0.0386,  -0.8571,   0.7265,   0.9405],
        [ -0.0386,  -0.0476,  -1.0572,   0.8961,   1.1601],
        [ -0.8571,  -1.0572, -23.4711,  19.8953,  25.7557],
        [  0.7265,   0.8961,  19.8953, -16.8643, -21.8319],
        [  0.9405,   1.1601,  25.7557, -21.8319, -28.2627]])
b tensor([  0.6044,   0.7455,  16.5508, -14.0294, -18.1619])
c tensor([-11.6710])
```

Parameter Update

NN App Algorithm Sketch

- Create a model and define a loss (i.e., **construct a computation graph**)
- For each example
  - **Forward: calculate the result** (prediction & loss) of that computation
  - if training
    - perform **back propagation**
    - **update** parameters

Optimizer Update

- Most deep learning toolkits implement the parameter updates by calling **optimizer.step()** function

```
[59] # Before gradient update
     for name, param in model.named_parameters():
         print(name, param)
```

```
A Parameter containing:
tensor([[-0.2267,  0.6521, -0.8193,  0.7723, -0.6456],
        [ 1.2410, -2.4380, -0.5612, -0.1144, -0.2687],
        [ 0.4792,  0.4543, -1.3530,  1.2934, -0.9943],
        [ 0.7565,  0.9449,  0.2796,  0.4703,  0.2926],
        [ 0.4143,  0.5891,  0.4370,  0.6060,  0.0161]]
b Parameter containing:
tensor([ 0.3592,  0.3455, -0.2517, -0.5678, -0.6016], :
c Parameter containing:
tensor([-1.5490], requires_grad=True)
```

Before optimizer update

```
[61] # Gradient update:
     optimizer.step()
     # After gradient update
     for name, param in model.named_parameters():
         print(name, param)
```

```
A Parameter containing:
tensor([[-0.2167,  0.6621, -0.8093,  0.7623, -0.6556],
        [ 1.2510, -2.4280, -0.5512, -0.1244, -0.2787],
        [ 0.4892,  0.4643, -1.3430,  1.2834, -1.0043],
        [ 0.7465,  0.9349,  0.2696,  0.4803,  0.3026],
        [ 0.4043,  0.5791,  0.4270,  0.6160,  0.0261]]
b Parameter containing:
tensor([ 0.3492,  0.3355, -0.2617, -0.5578, -0.5916], :
c Parameter containing:
tensor([-1.5390], requires_grad=True)
```

After optimizer update

Many Different Optimizers

- **Simple SGD:** update with only gradients
- **Momentum:** update w/ running average of gradient
- **Adagrad:** update downweighting high-variance values
- **Adam:** update w/ running average of gradient, downweighting by running average of variance

Standard SGD

- **Reminder:** Standard stochastic gradient descent does

$$g_t = \nabla_{\theta_{t-1}} \ell(\theta_{t-1})$$

Gradient of Loss

$$\theta_t = \theta_{t-1} - \eta g_t$$

Learning Rate

- There are many other optimization options! (see Ruder 2016 in references)

SGD With Momentum

- Remember gradients from past time steps

$$v_t = \gamma v_{t-1} + \eta g_t$$

Momentum

Previous Momentum

Momentum Conservation Parameter

$$\theta_t = \theta_{t-1} - v_t$$

- **Intuition:** Prevent instability resulting from sudden changes

Adagrad

- Adaptively reduce learning rate based on accumulated variance of the gradients

$$G_t = G_{t-1} + g_t \odot g_t$$

Squared Current Gradient

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

Small Constant

- **Intuition:** frequently updated parameters (e.g. common word embeddings) should be updated less
- **Problem:** learning rate continuously decreases, and training can stall -- fixed by using rolling average in *AdaDelta* and *RMSProp*

Adam

- Most standard optimization option in NLP and beyond
- Considers rolling average of gradient, and momentum

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$ Momentum

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t \odot g_t$$ Rolling Average of Gradient

- Correction of bias early in training, because $\mathbb{E}[v_t]$ is a biased estimation of variance: $\mathbb{E}[v_t] = \mathbb{E}[g_t^2] \cdot (1 - \beta_2^t)$

$$\hat{m}_t = \frac{m_t}{1 - (\beta_1)^t} \quad \hat{v}_t = \frac{v_t}{1 - (\beta_2)^t}$$

- Final update

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Training Tricks

Shuffling the Training Data

- Stochastic gradient methods update the parameters a little bit at a time
    - What if we have the sentence “I love this sentence so much!” at the end of the training data 50 times?

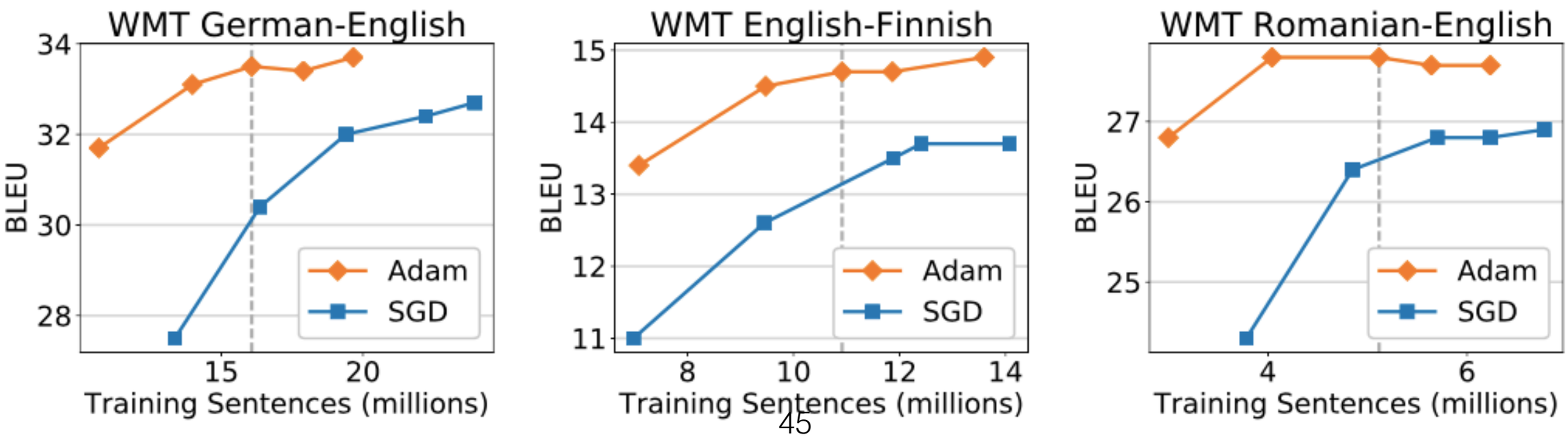
- To train correctly, we should randomly shuffle the order at each time step

Simple Methods to Prevent Over-fitting

- Neural nets have tons of parameters: we want to prevent them from over-fitting
- **Early stopping:**
  - monitor performance on held-out development data and stop training when it starts to get worse
- **Learning rate decay:**
  - gradually reduce learning rate as training continues, or
  - reduce learning rate when dev performance plateaus
- **Patience:**
  - learning can be unstable, so sometimes avoid stopping or decay until the dev performance gets worse *n* times

Which One to Use?

- Adam is usually fast to converge and stable
- But simple SGD tends to do very will in terms of generalization (Wilson et al. 2017)
- You should use learning rate decay, (e.g. on Machine translation results by Denkowski & Neubig 2017)



Dropout
(Srivastava+ 14)

- Neural nets have lots of parameters, and are prone to overfitting
- Dropout: randomly zero-out nodes in the hidden layer with probability *p* at **training time only**

- Because the number of nodes at training/test is different, scaling is necessary:
  - **Standard dropout:** scale by 1-*p* at test time
  - **Inverted dropout:** scale by 1/(1-*p*) at training time
- An alternative: **DropConnect** (Wan+ 2013) instead zeros out weights in the NN

Efficiency Tricks:

Operation Batching

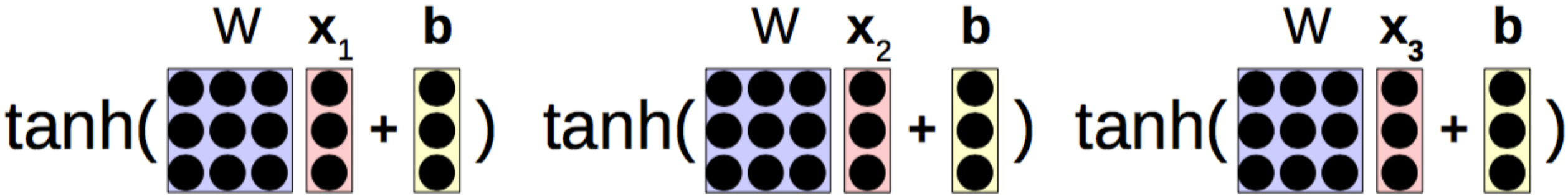
Efficiency Tricks: Mini-batching

- On modern hardware 10 operations of size 1 is **much slower than** 1 operation of size 10

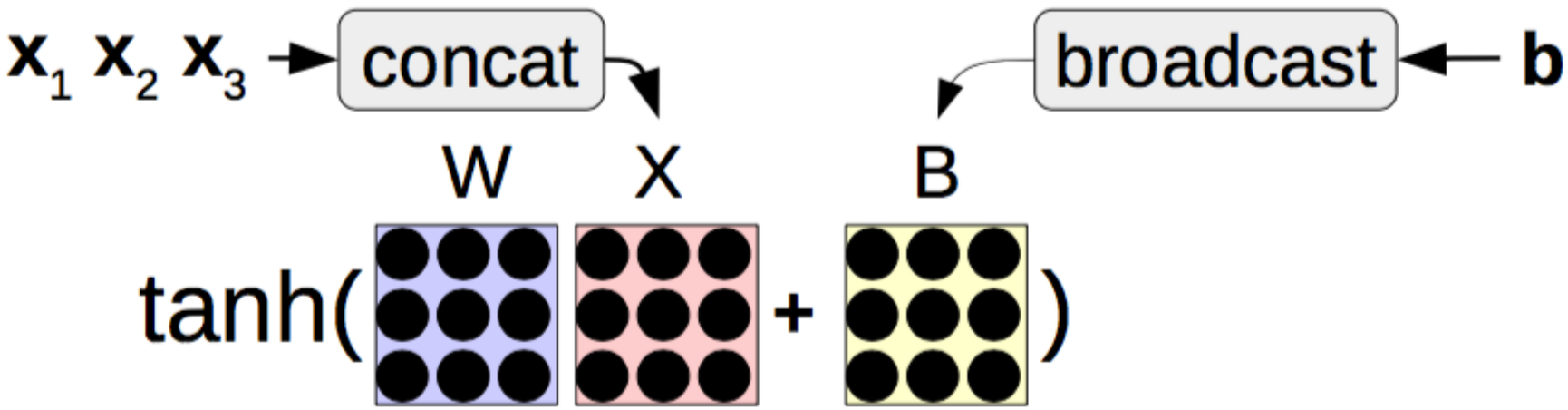
- Minibatching combines together smaller operations into one big one

Minibatching

Operations w/o Minibatching



Operations with Minibatching



Procedure of Minibatching

- **Group together similar operations** (e.g. loss calculations for a single word) and execute them all together
  - In the case of a feed-forward language model, each word prediction in a sentence can be batched
  - For recurrent neural nets, etc., more complicated
- How this works depends on toolkit
  - Most toolkits have require you to **add an extra dimension** representing the batch size
  - Some toolkits have **explicit tools** that help with batching

Assignment

Still Some Things Left!

- We've left off the details of some underlying parts.
- What about more operations?
- What about more optimizers?
- **Challenge**: can you make a more sophisticated model?

https://github.com/JunjieHu/cs769-assignments/tree/main/assignment1

load_embedding

- Load a pretrained word embedding matrix from a text file (GloVE/ FastText)

```
def load_embedding(vocab, emb_file, emb_size):
    """
    Read embeddings for words in the vocabulary from the emb_file (e.g., GloVe, FastText).
    Args:
        vocab: (Vocab), a word vocabulary
        emb_file: (string), the path to the embdding file for loading
        emb_size: (int), the embedding size (e.g., 300, 100) depending on emb_file
    Return:
        emb: (np.array), embedding matrix of size (|vocab|, emb_size)
    """
    raise NotImplementedError()
```

- Copy the numpy embedding matrix to torch embeddings

```
class DanModel(BaseModel):
    def copy_embedding_from_numpy(self):
        """
        Load pre-trained word embeddings from numpy.array to nn.embedding
        Pass hyperparameters explicitly or use self.args to access the hyperparameters.
        """
        raise NotImplementedError()
```

DanModel

```python
class DanModel(BaseModel):
    def __init__(self, args, vocab, tag_size):
        super(DanModel, self).__init__(args, vocab, tag_size)
        self.define_model_parameters()
        self.init_model_parameters()

        # Use pre-trained word embeddings if emb_file exists
        if args.emb_file is not None:
            self.copy_embedding_from_numpy()

    def define_model_parameters(self):
        """
        Define the model's parameters, e.g., embedding layer, feedforward layer.
        Pass hyperparameters explicitly or use self.args to access the hyperparameters.
        """
        raise NotImplementedError()

    def init_model_parameters(self):
        """
        Initialize the model's parameters by uniform sampling from a range [-v, v], e.g., v=0.08
        Pass hyperparameters explicitly or use self.args to access the hyperparameters.
        """
        raise NotImplementedError()

    def copy_embedding_from_numpy(self):
        """
        Load pre-trained word embeddings from numpy.array to nn.embedding
        Pass hyperparameters explicitly or use self.args to access the hyperparameters.
        """
        raise NotImplementedError()

    def forward(self, x):
        """
        Compute the unnormalized scores for P(Y|X) before the softmax function.
        E.g., feature: h = f(x)
              scores: scores = w * h + b
              P(Y|X) = softmax(scores)
```

Mini-batch of varying length sequences

Add padding tokens to the end of the sentences to make sure that the minibatch has all sentences of the same length

Example: The maximum length is 5.

- X1 = “ I love this interesting movie” // 5 real-words
- X2 = “ great movie [pad] [pad] [pad]” // 2 real-words
- X3 = “This movie is bad [pad] ” // 4 real-words

When computing the average word embedding, we should (1) zero-out the padding tokens, (2) sum out the embeddings over words, and (3) divide the sum by the number of real-words

Questions?