CS11-711 Advanced NLP

Building a Neural Network Toolkit for NLP

`minnn`

Graham Neubig

Carnegie Mellon University
Language Technologies Institute

Site
https://phontron.com/class/anlp2021/

Neural Network Frameworks

theano

∂y/net

Caffe

Chainer

mxnet

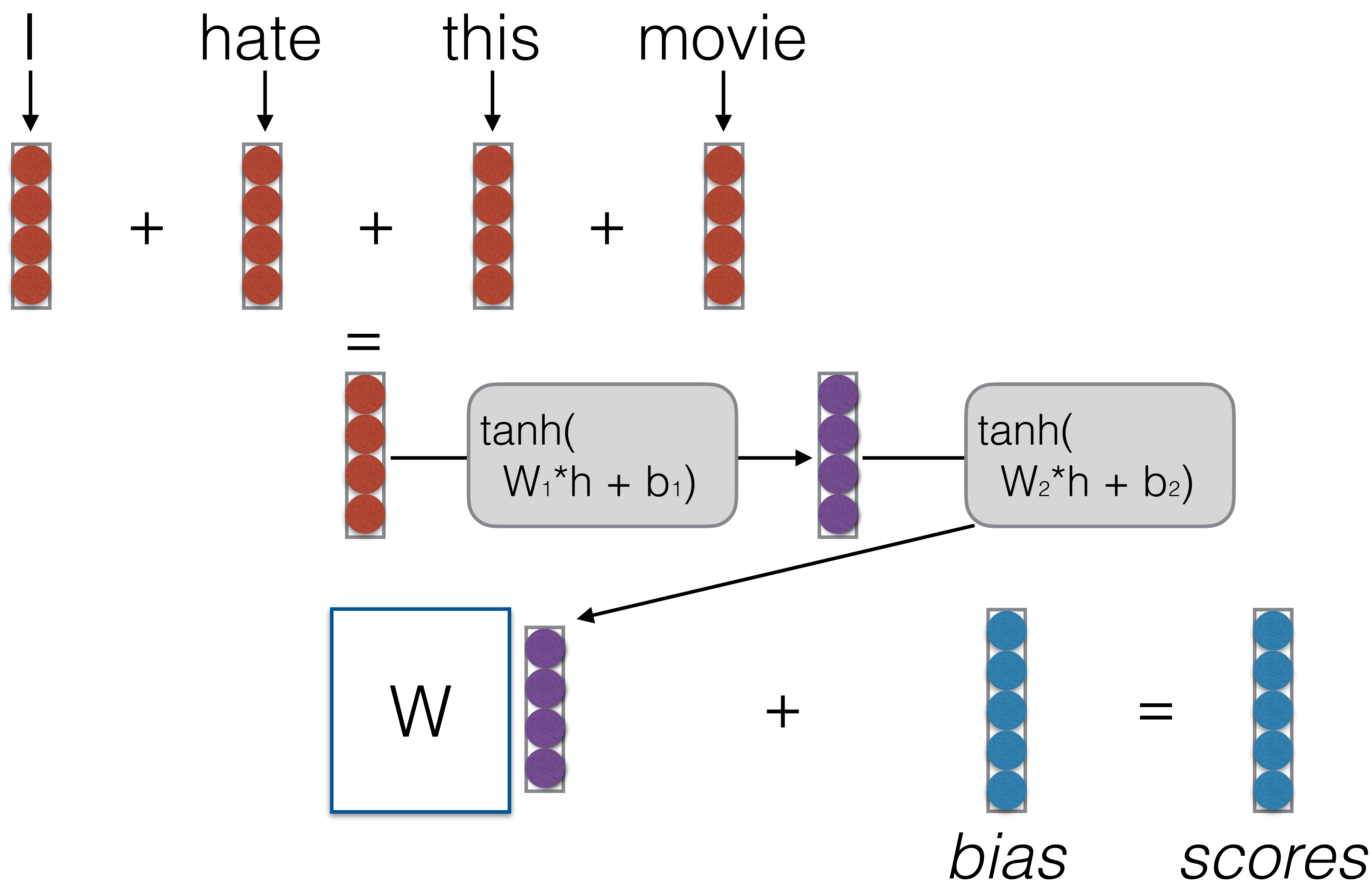
PYTORCH

TensorFlow

JAX

minnn

Example App: Deep CBOW Model



Algorithm Sketch for NN App Code

- Create a model
- For each example
  - **create a graph** that represents the computation you want
  - **calculate the result** 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, custom-written kernels
- **minnn:** numpy/CuPy

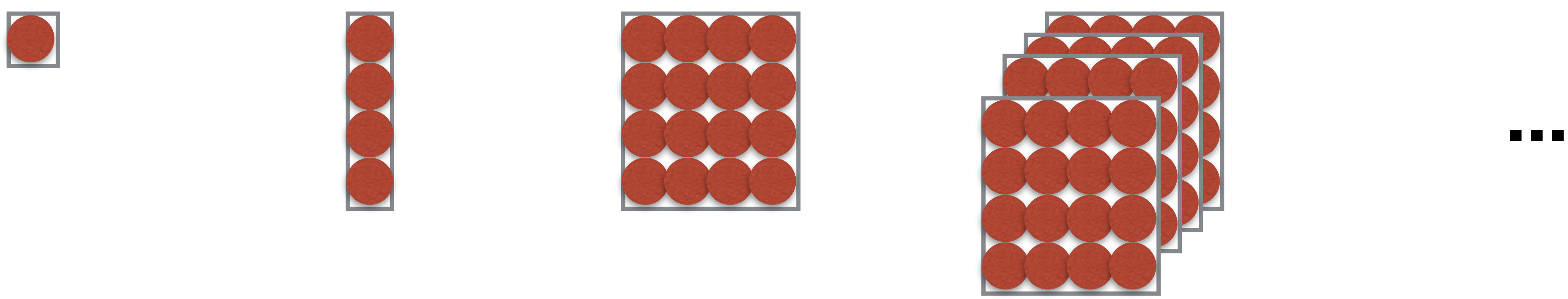
```
import numpy as np

a = [[1, 0], [0, 1]]
b = [[4, 1], [2, 2]]
np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

- Many many different operations
- CuPy is a clone of NumPy that works on GPU

Tensors

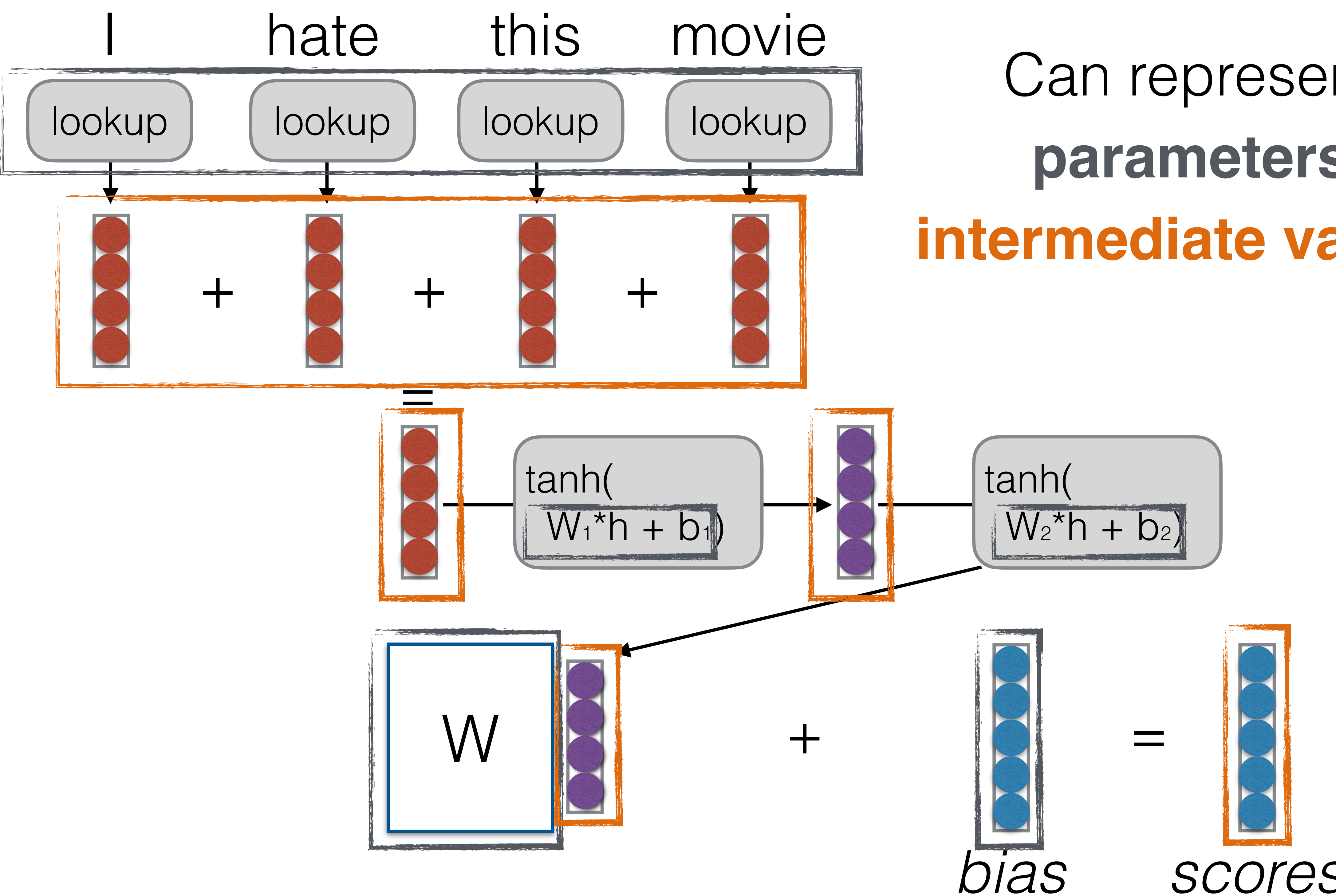
- An n-dimensional array

<u>Scalar</u> **<u>Vector</u>** **<u>Matrix</u>** **<u>3-dim Tensor</u>**



- Widely used in neural networks
- Implementation in minnn saves both values and gradients

Tensors in Neural Networks



Can represent
parameters
intermediate values

Tensor Data Structure Definition

```
# Tensor
class Tensor:
  def __init__(self, data: xp.ndarray):
    self.data: xp.ndarray = data

    # gradient, should be the same size as data
    self.grad: Union[Dict[int, xp.ndarray], xp.ndarray] = None

    # generated from which operation?
    self.op: Op = None
```

Model and Parameter Definition

Algorithm Sketch

- Create a model
- For each example
  - **create a graph** that represents the computation you want
  - **calculate the result** of that computation
  - if training
    - perform **back propagation**
    - **update** parameters

Example Model Creation (in App Code)

```
# Define the model
EMB_SIZE = args.emb_size
HID_SIZE = args.hid_size
HID_LAY = args.hid_layer

W_emb = model.add_parameters((nwords, EMB_SIZE))

W_h = [model.add_parameters(
       (HID_SIZE, EMB_SIZE if lay == 0 else HID_SIZE),
          initializer='xavier_uniform')
             for lay in range(HID_LAY)]

W_sm = model.add_parameters((ntags, HID_SIZE),
                  initializer='xavier_uniform')
```

Model Class, Adding Parameters

```
# Model: collection of parameters
class Model:
    def __init__(self):
        self.params: List[Parameter] = []

    def add_parameters(self, shape,
                       initializer='normal',
                       **initializer_kwargs):
        init_f = getattr(Initializer, initializer)
        data = init_f(shape, **initializer_kwargs)
        param = Parameter(data)
        self.params.append(param)
        return param
```

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 [-0.1, 0.1] 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: $$\sqrt{\frac{6}{d_{in} + d_{out}}}$$

Computation Definition

NN App Algorithm Sketch

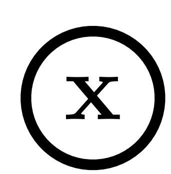
Greedy Computation
(cf Lazy Computation)

- Create a model
- For each example
  - **create a graph** that represents the computation you want
  - **calculate the result** of that computation
  - if training
    - perform **back propagation**
    - **update** parameters

expression:

$$\mathbf{x}$$

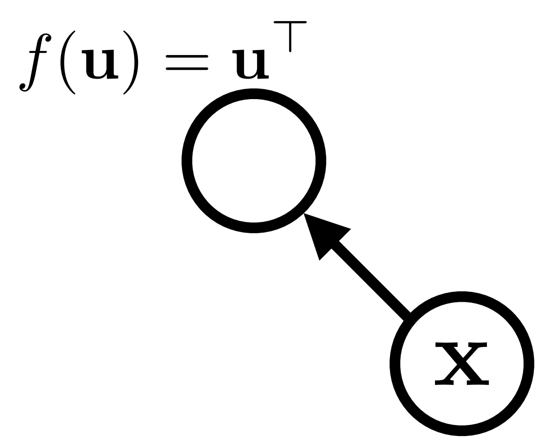
graph:



expression:

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

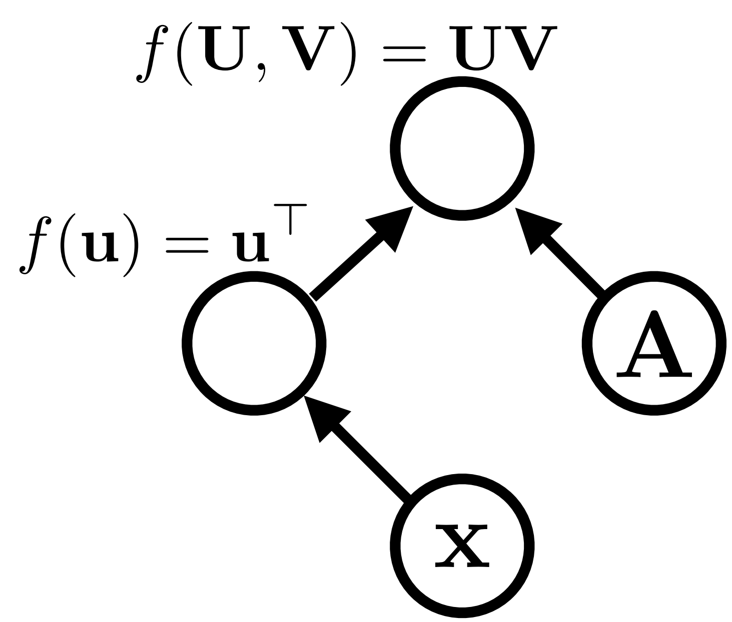
graph:



expression:

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

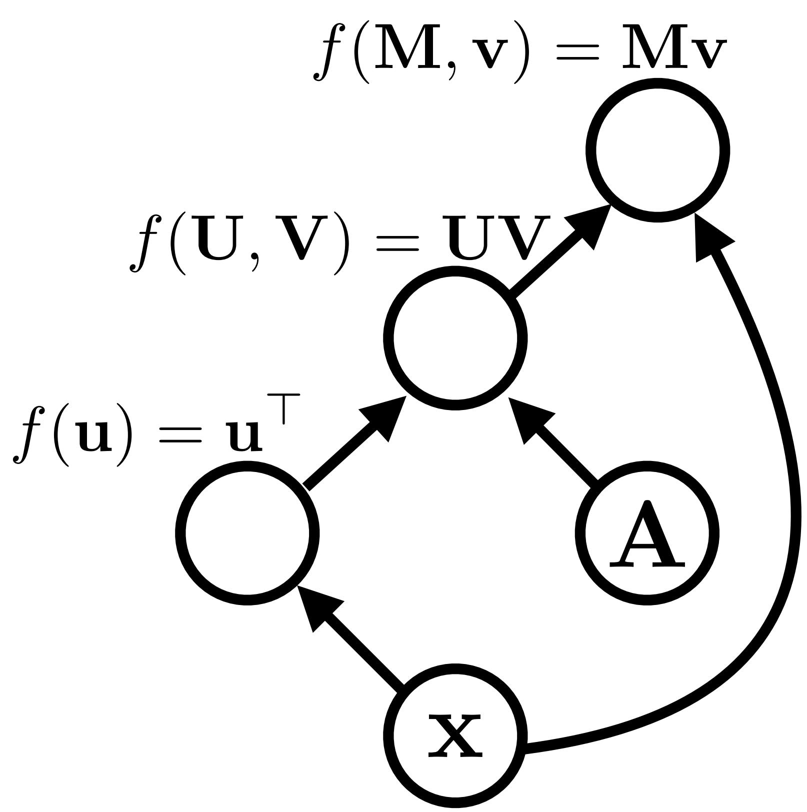
graph:



expression:

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

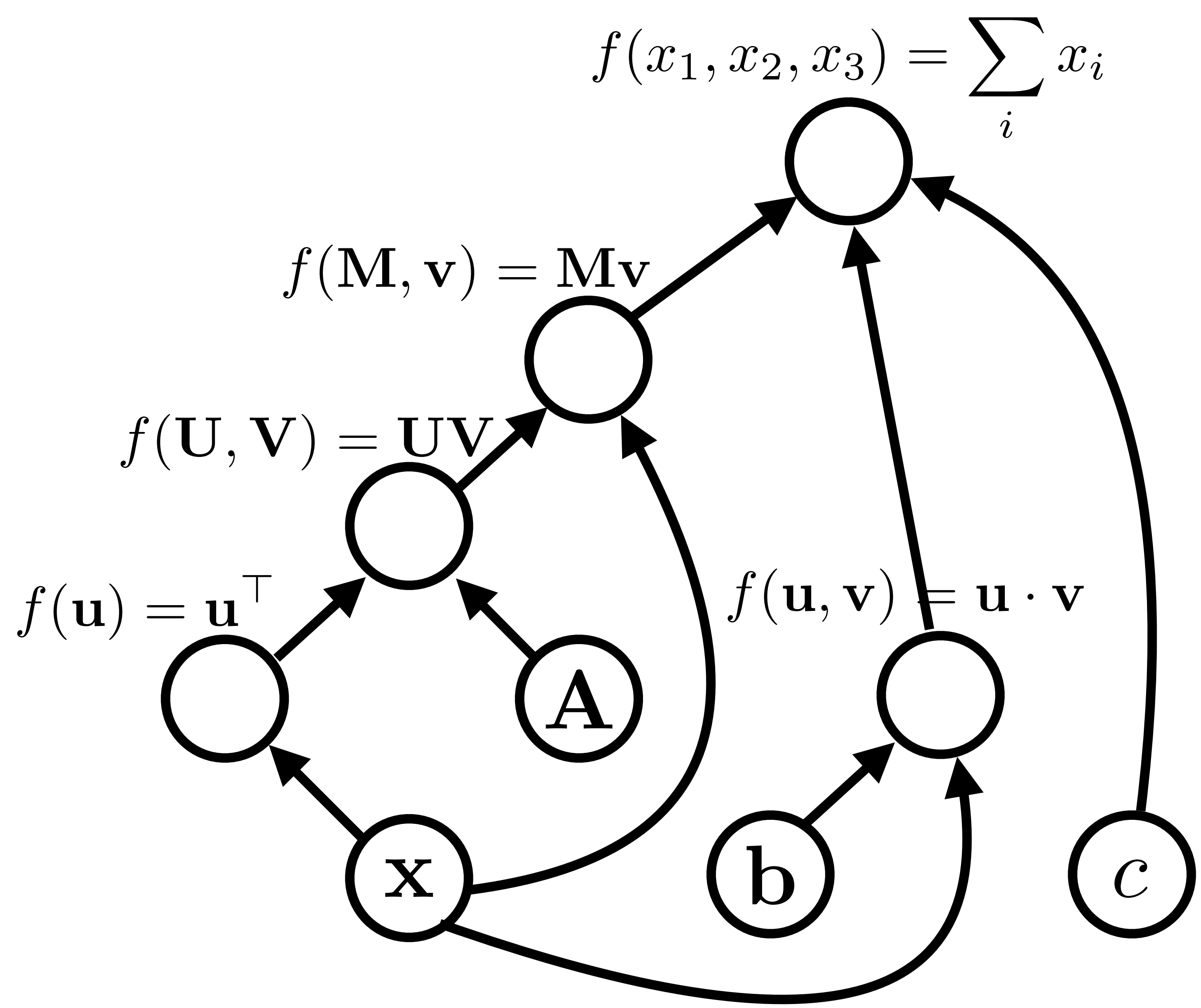
graph:



expression:

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

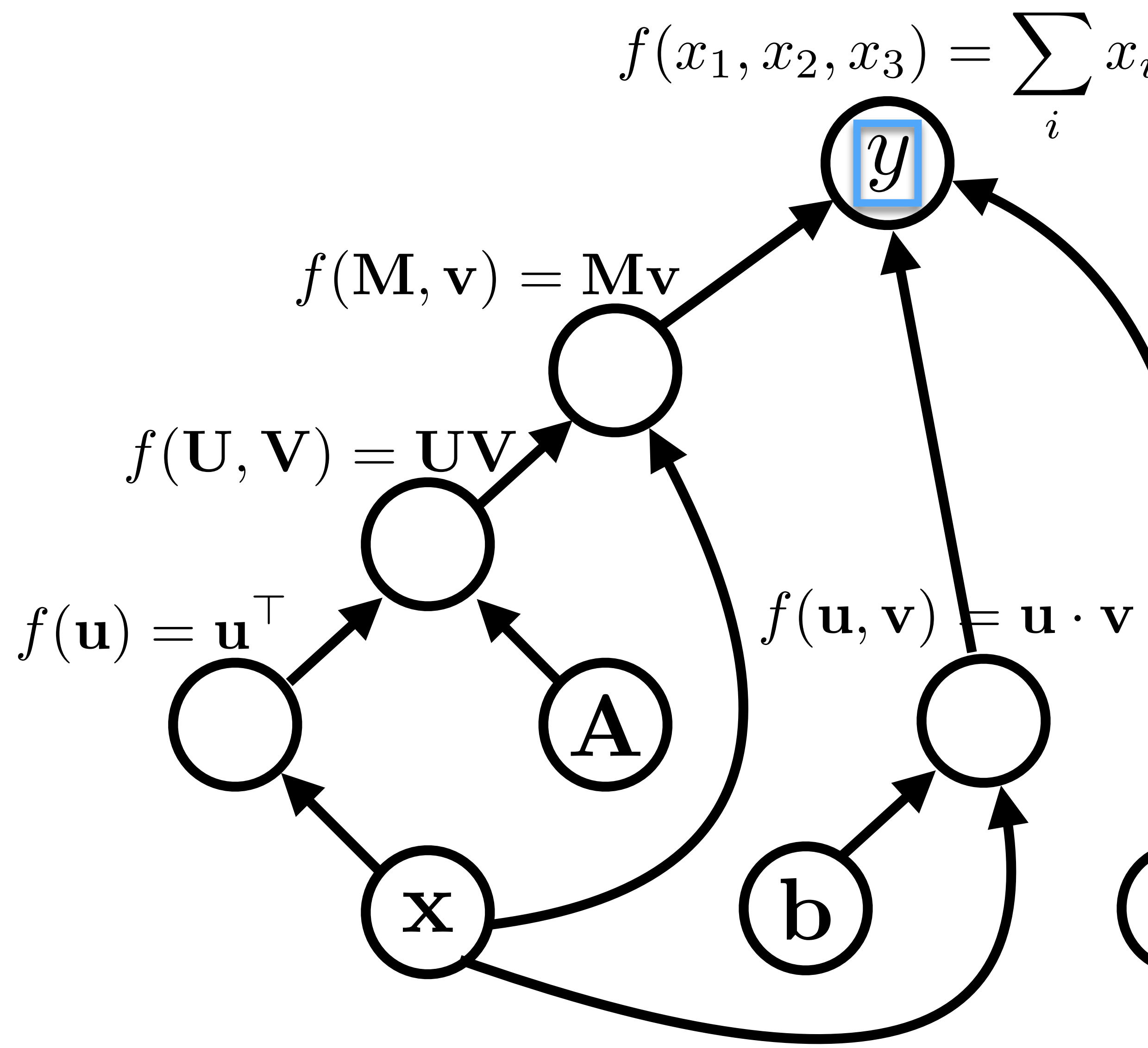
graph:



expression:

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

graph:



Example Graph Creation (in App Code)

```
mn.reset_computation_graph()

emb = mn.lookup(W_emb, words)
h = mn.sum(emb, axis=0)
for W_h_i, b_h_i in zip(W_h, b_h):
    h = mn.tanh(mn.dot(W_h_i, h) + b_h_i)
return mn.dot(W_sm, h) + b_sm
```

Computation Graph

```python
class ComputationGraph:
    # global cg
    _cg: 'ComputationGraph' = None

    @classmethod
    def get_cg(cls, reset=False):
        if ComputationGraph._cg is None or reset:
            ComputationGraph._cg = ComputationGraph()
        return ComputationGraph._cg

    def __init__(self):
        self.ops: List[Op] = []

    def reg_op(self, op: Op):
        assert op.idx is None
        op.idx = len(self.ops)
        self.ops.append(op)
```

Operations

- Operations must know:
- **Forward:** how to calculate their value given input

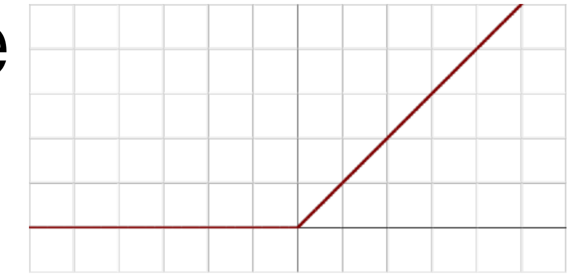
$$f(\mathbf{u})$$

- **Backward:** how to calculate their derivative given following derivative

$$\frac{\partial f(\mathbf{u})}{\partial \mathbf{u}} \frac{\partial \mathcal{F}}{\partial f(\mathbf{u})}$$

Example Op: Relu

Value



Gradient

```
class OpRelu(Op):
    def forward(self, t: Tensor):
        arr_relu = t.data
        arr_relu[arr_relu < 0.0] = 0.0
        t_relu = Tensor(arr_relu)
        self.store_ctx(t=t, t_relu=t_relu, arr_relu=arr_relu)
        return t_relu

    def backward(self):
        t, t_relu, arr_relu = self.get_ctx('t', 't_relu', 'arr_relu')
        if t_relu.grad is not None:
            grad_t = xp.where(arr_relu > 0.0, 1.0, 0.0) * t_relu.grad
            t.accumulate_grad(grad_t)

def relu(param): return OpRelu().full_forward(param)
```

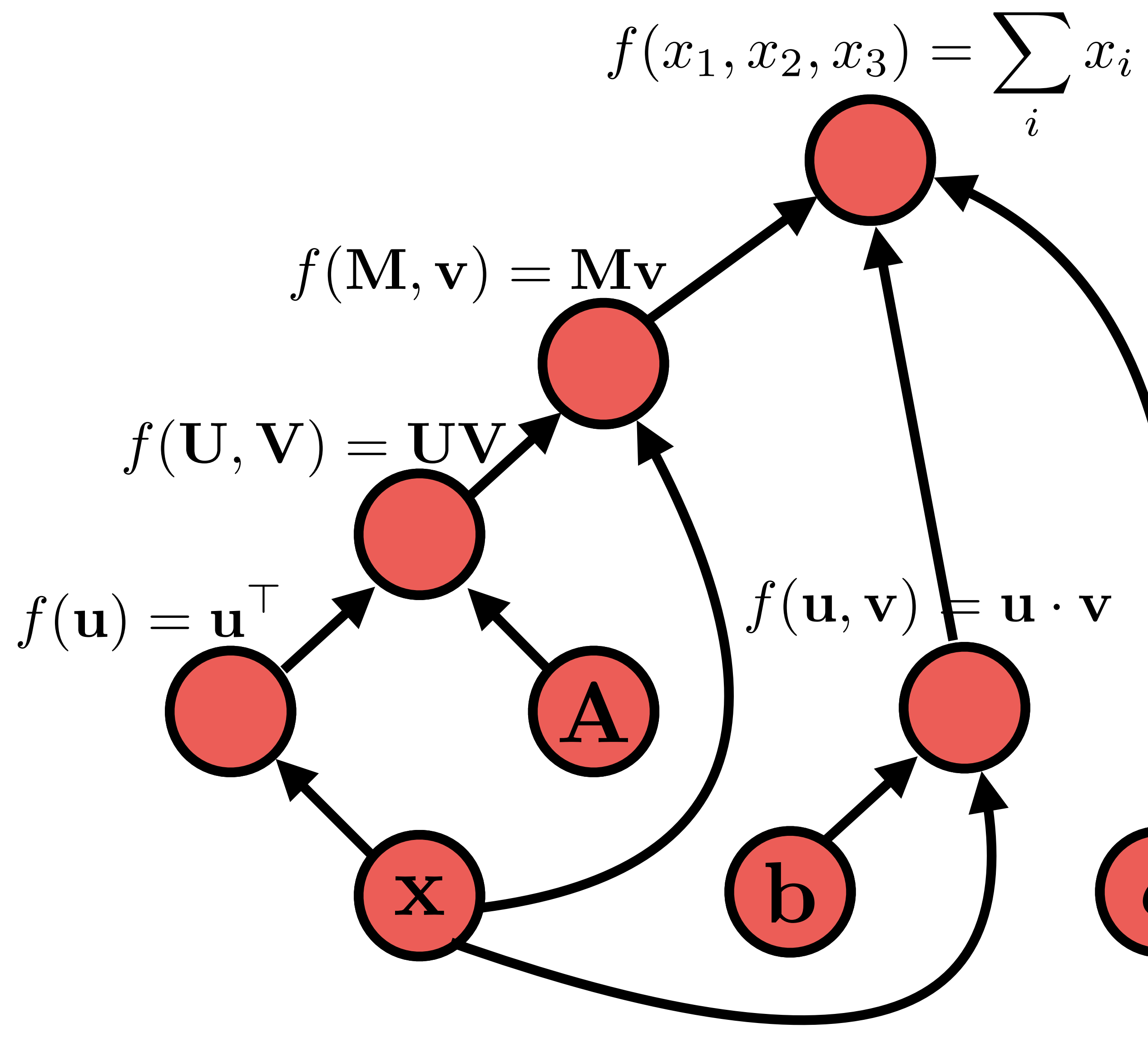
Back Propagation

NN App Algorithm Sketch

- Create a model
- For each example
  - **create a graph** that represents the computation you want
  - **calculate the result** of that computation
  - if training
    - perform **back propagation**
    - **update** parameters

Back Propagation

graph:



Backward Code

```python
def backward(t: Tensor, alpha=1.):
    # first put grad to the start one
    t.accumulate_grad(alpha)
    # locate the op
    op = t.op
    # backward the whole graph!!
    cg = ComputationGraph.get_cg()
    for idx in reversed(range(op.idx+1)):
        cg.ops[idx].backward()
```

Parameter Update

NN App Algorithm Sketch

- Create a model
- For each example
  - **create a graph** that represents the computation you want
  - **calculate the result** of that computation
  - if training
    - perform **back propagation**
    - **update** parameters

Many Different Update Rules

- **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

$$\underbrace{g_t = \nabla_{\theta_{t-1}} \ell(\theta_{t-1})}_{\text{Gradient of Loss}}$$

$$\theta_t = \theta_{t-1} - \underbrace{\eta}_{\text{Learning Rate}} g_t$$

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

SGD Update Rule

```
class SGDTrainer(Trainer):
    def __init__(self, model: Model, lrate=0.1):
        super().__init__(model)
        self.lrate = lrate

    def update(self):
        lrate = self.lrate
        for p in self.model.params:
            if p.grad is not None:
                if isinstance(p.grad, dict):  # sparsely update to save time!
                    self.update_sparse(p, p.grad, lrate)
                else:
                    self.update_dense(p, p.grad, lrate)
            # clean grad
            p.grad = None

    def update_dense(self, p: Parameter, g: xp.ndarray, lrate: float):
        p.data -= lrate * g

    def update_sparse(self, p: Parameter,
                      gs: Dict[int, xp.ndarray], lrate: float):
        for widx, arr in gs.items():
            p.data[widx] -= lrate * arr
```

SGD With Momentum

- Remember gradients from past time steps

$$\underbrace{v_t}_{\text{Momentum}} = \underbrace{\gamma}_{\text{Momentum Conservation Parameter}} \underbrace{v_{t-1}}_{\text{Previous Momentum}} + \eta g_t$$

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

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

Adagrad

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

$$G_t = G_{t-1} + \underbrace{g_t \odot g_t}_{\text{Squared Current Gradient}}$$

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

— 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

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

- Final update

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

Training Tricks

Shuffling the Training Data

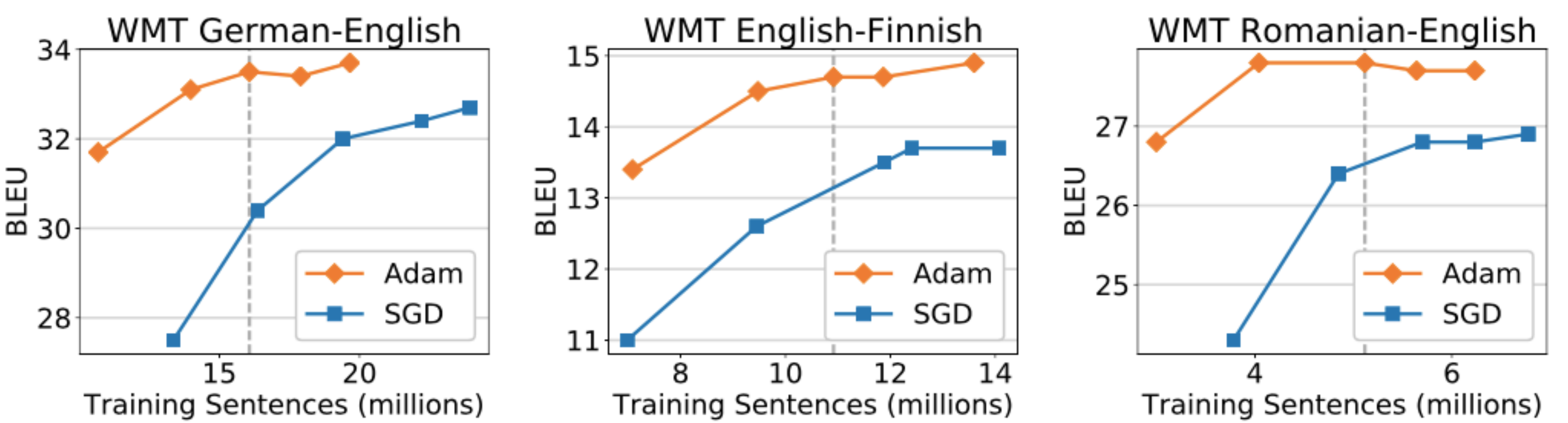
- Stochastic gradient methods update the parameters a little bit at a time
  - What if we have the sentence “I love this sentence so much!” at the end of the training data 50 times?
- To train correctly, we should randomly shuffle the order at each time step

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?

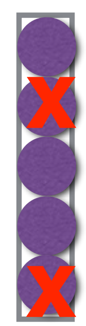
- Adam is usually fast to converge and stable
- But simple SGD tends to do very will in terms of generalization (Wilson et al. 2017)
- You should use learning rate decay, (e.g. on Machine translation results by Denkowski & Neubig 2017)



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**



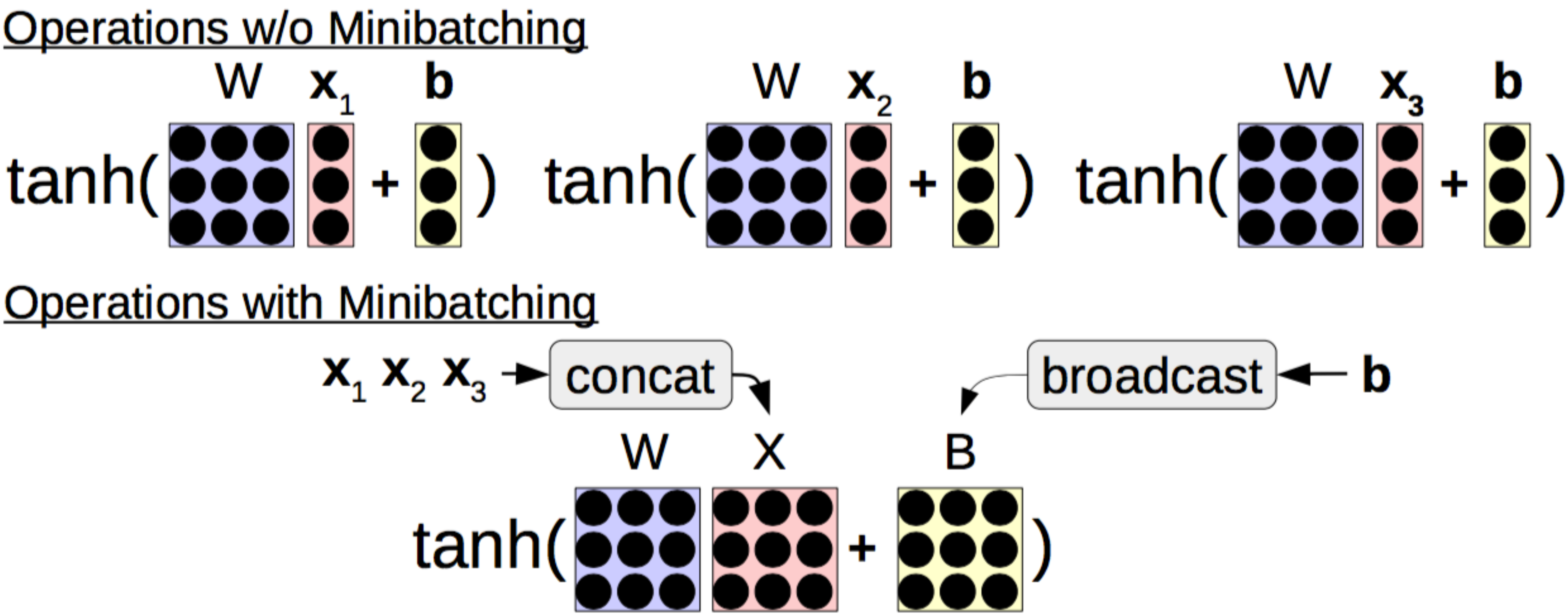
- Because the number of nodes at training/test is different, scaling is necessary:
  - **Standard dropout:** scale by *p* at test time
  - **Inverted dropout:** scale by 1/(1-*p*) at training time
- An alternative: **DropConnect** (Wan+ 2013) instead zeros out weights in the NN

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



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/neubig/minnn-assignment/

Questions?