CS378: Natural Language Processing

Lecture 4: Feedforward Neural Network

TEXAS
The University of Texas at Austin

Eunsol Choi

Slides adapted from Greg Durrett, Yoav Artzi, Yejin Choi, Princeton NLP course

Logistics

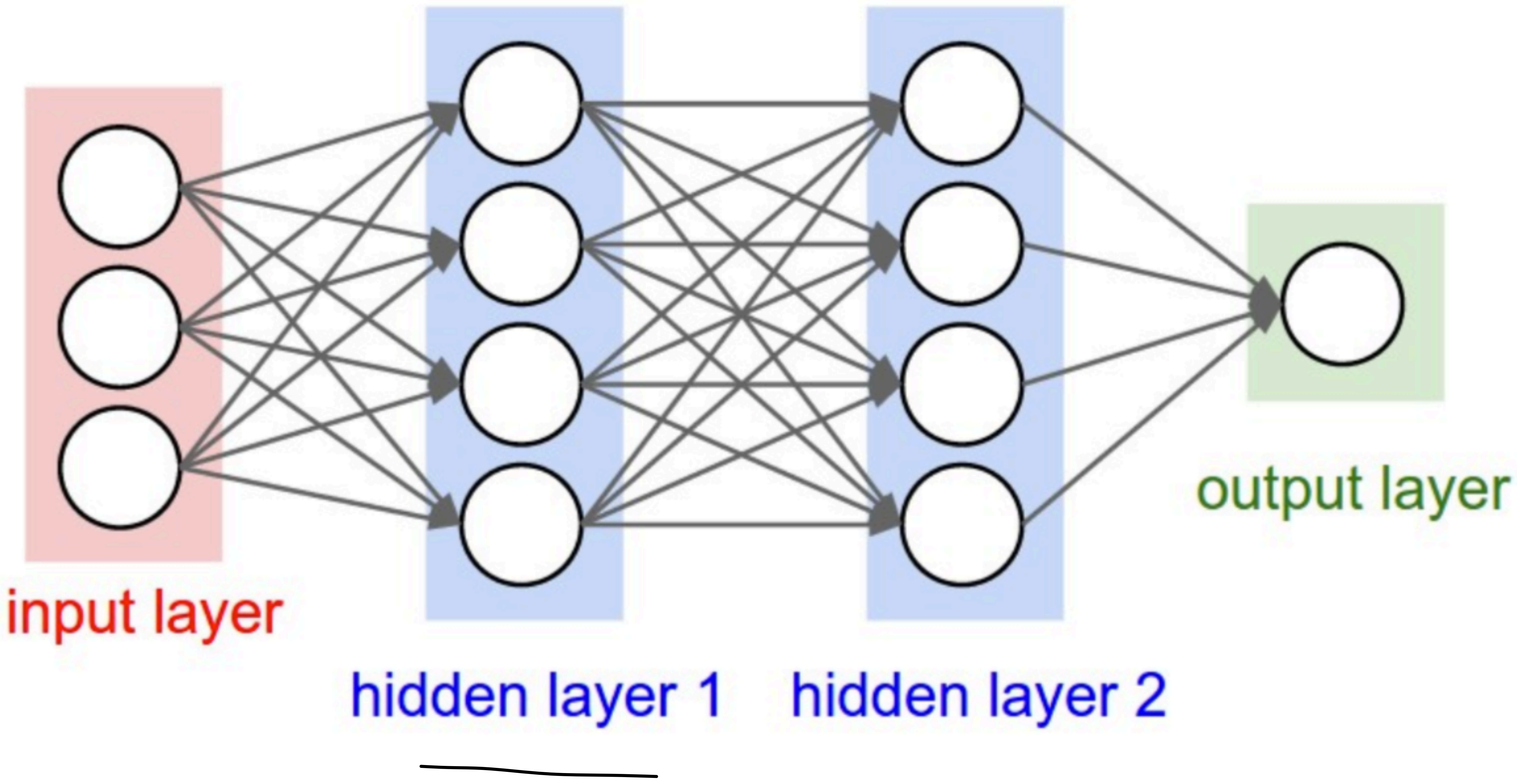
- Final project is released!
- Please feel free to post clarification questions!
  - The final project guideline covers many concepts not yet introduced in the course, don't worry!
  - But it might be useful to set up GPU and try learning the starter code when you have time.
  - It will become a lot clearer once we get to Week 7-8!

Today

- Introduction to neural network
- Introduction to computational graph
- Introduction to backpropagation
- Few practical tips..
  - training neural network
  - PyTorch introduction

A neural network

- If we feed inputs through multiple logistic regression functions, then we can construct a output vector…
- which we can feed into another logistic regression function

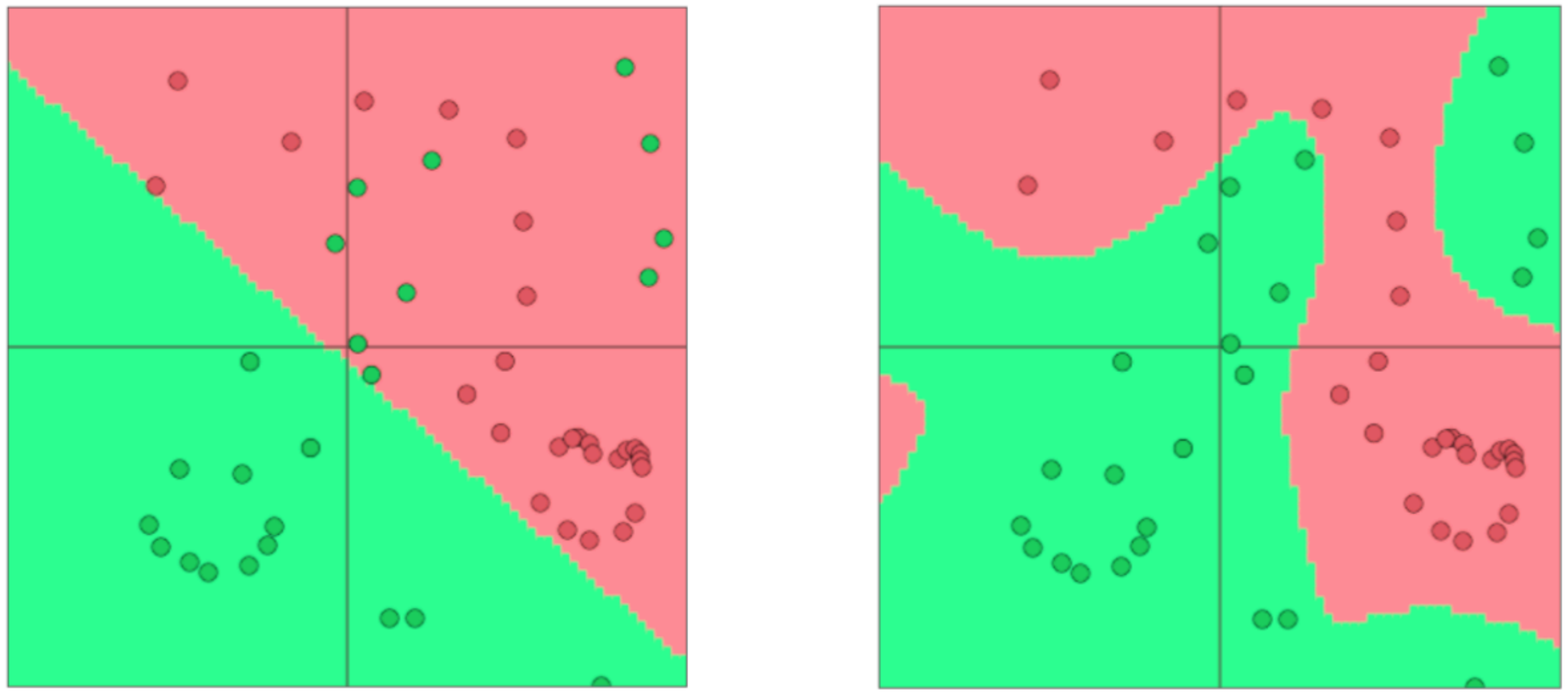


$$P(\mathbf{y} \mid \mathbf{x}) = \text{softmax}(Wg(H^2(g(H^1 f(\mathbf{x})))))$$

$$P(\mathbf{y} \mid \mathbf{x}) = \text{softmax}(Wg(H^1 f(\mathbf{x})))$$

Recap: The "Promise"

- Deep Learning: attempts to learn multilevel of representation of increasing complexity / abstraction



Recap: The "Promise"

- Representation Learning: automatically learn good features and representations
  - MaxEnt (multinomial logistic regression):

$$P(\mathbf{y} \mid \mathbf{x}) = \text{softmax}(w \cdot \underline{\phi(x, y)})$$

You design the feature vector

  - NNs:

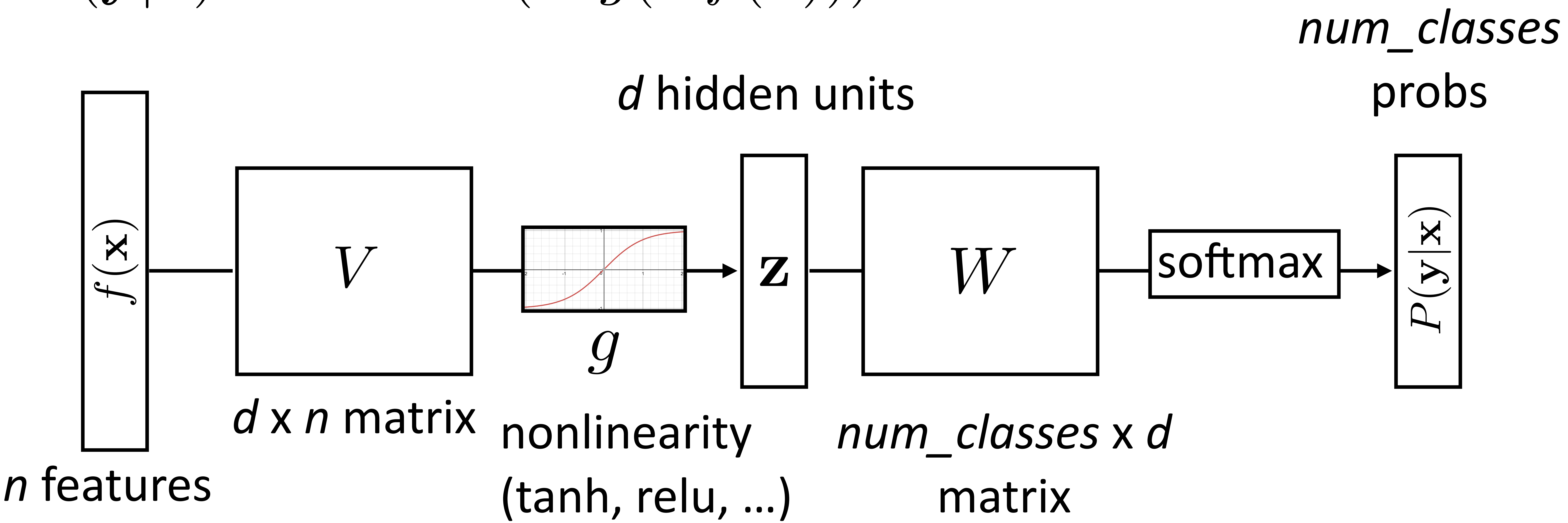
$$P(\mathbf{y} \mid \mathbf{x}) = \text{softmax}(W\underline{g(H^1 f(\mathbf{x}))})$$

$$P(\mathbf{y} \mid \mathbf{x}) = \text{softmax}(W\underline{g(H^2(g(H^1 f(\mathbf{x}))))})$$

Feature representations are "learned" through hidden layers

Neural Networks for Classification

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



Learning Objective

- We will use the same log likelihood objective we used for log linear models for classification tasks

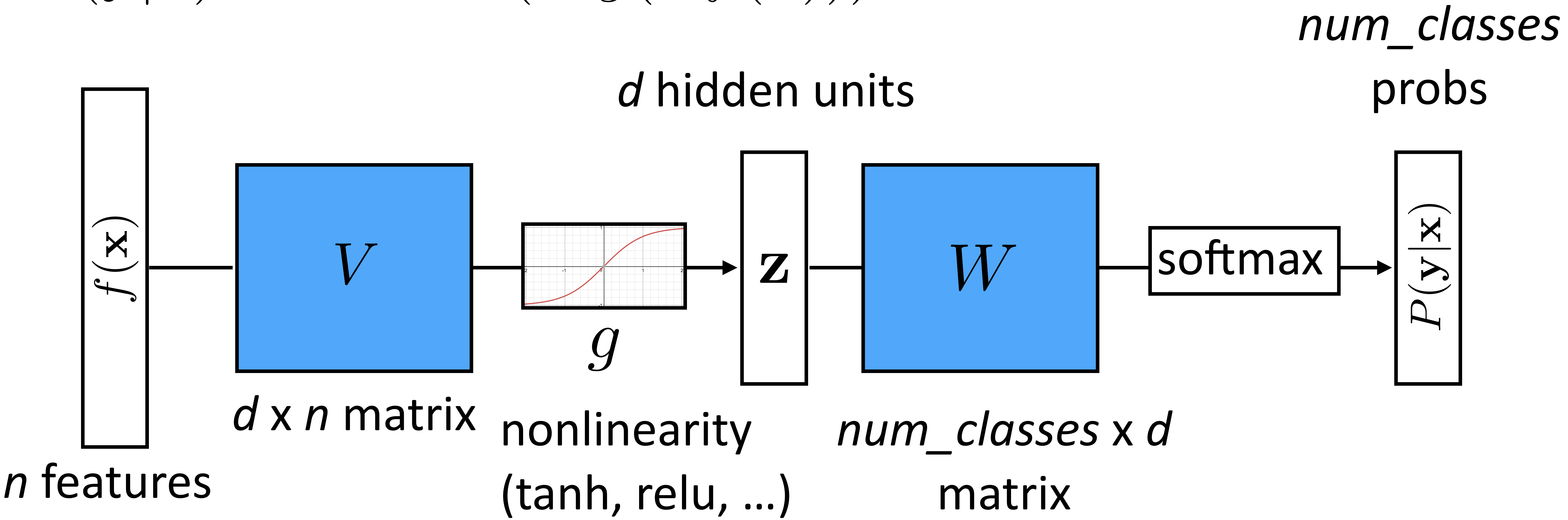
$$L(w) = \log \prod_{i=1}^{N} p(y^i \,|\, x^i; w) = \sum_{i=1}^{N} \log(p(y^i \,|\, x^i; w))$$

$$w* = \text{argmax}_w L(w)$$

- How do we compute the derivative, when there are multiple layers of parameters?

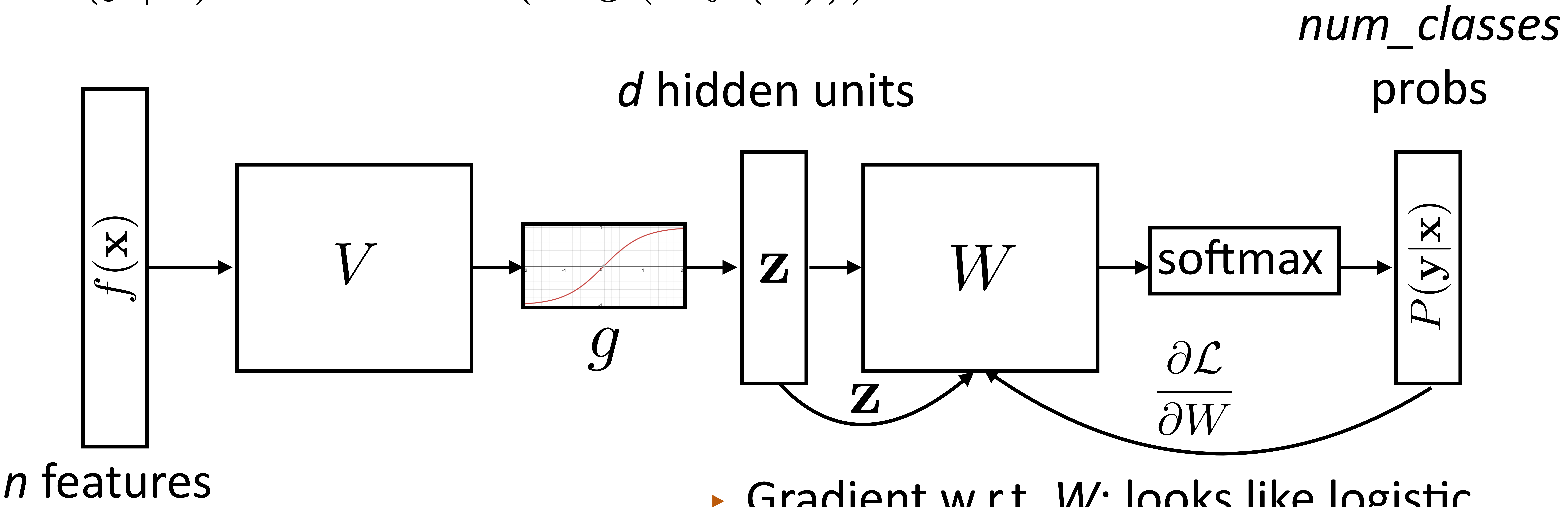
Neural Networks for Classification

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



Computing Gradients: Step 1

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



- Gradient w.r.t. *W*: looks like logistic regression, can be computed treating **z** as the input features

Computing Gradients: Step 1

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W\mathbf{z}) \qquad \mathbf{z} = g(Vf(\mathbf{x}))$$

- Maximize log likelihood of training data

$$\mathcal{L}(\mathbf{x}, i^*) = \log P(y = i^*|\mathbf{x}) = \log\left(\text{softmax}(W\mathbf{z}) \cdot e_{i^*}\right)$$

- *i**: index of the gold label
- e_i: num_classes dimension vector. 1 in the *i*th row, zero elsewhere.

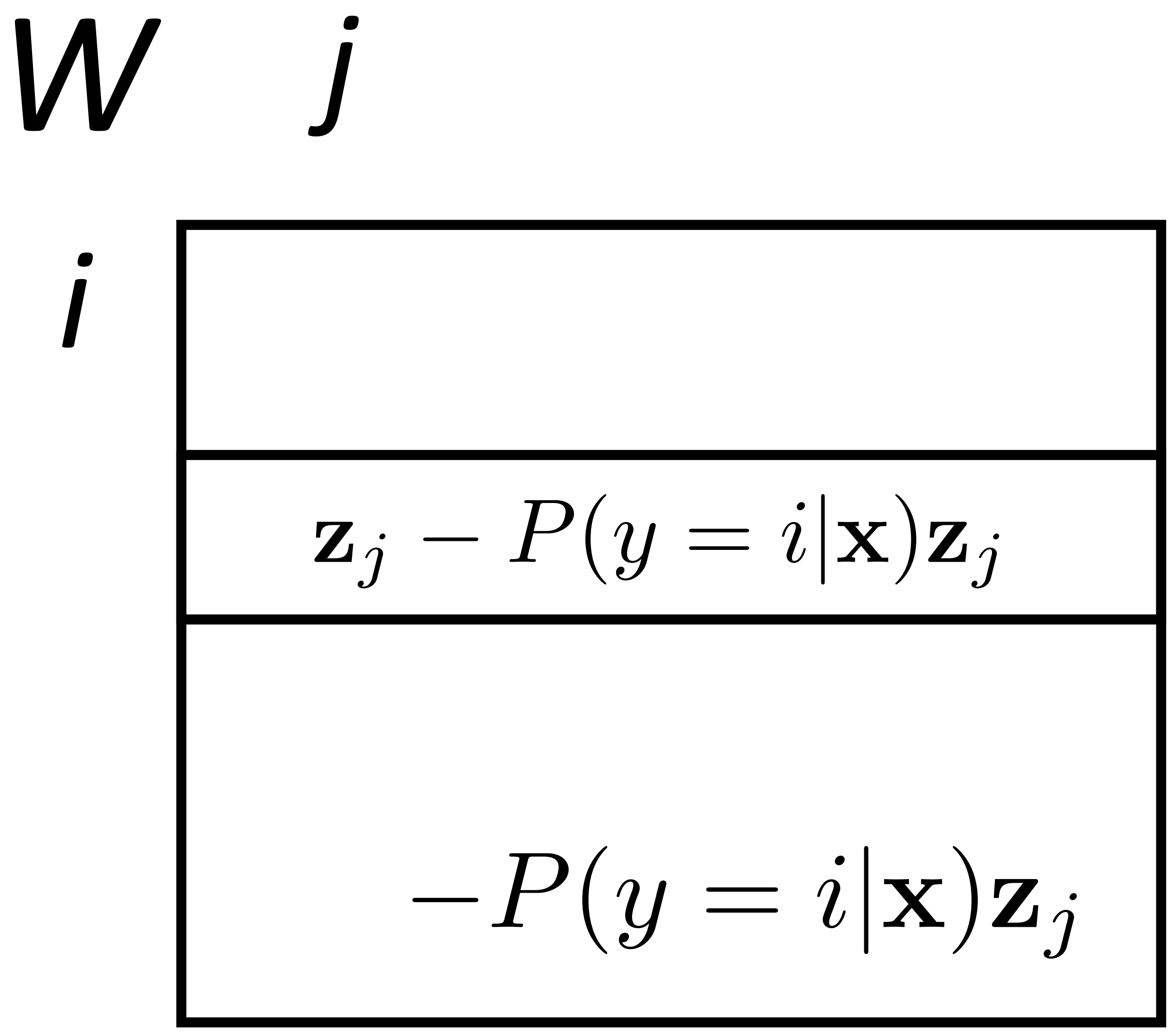
$$\mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W\mathbf{z}) \cdot e_j$$

Computing Gradients: Step 1

$$\mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W\mathbf{z}) \cdot e_j$$

- Gradient with respect to *W*

$$\frac{\partial}{\partial W_{ij}} \mathcal{L}(\mathbf{x}, i^*) = \begin{cases} \mathbf{z}_j - P(y = i|\mathbf{x})\mathbf{z}_j & \text{if } i = i^* \\ -P(y = i|\mathbf{x})\mathbf{z}_j & \text{otherwise} \end{cases}$$

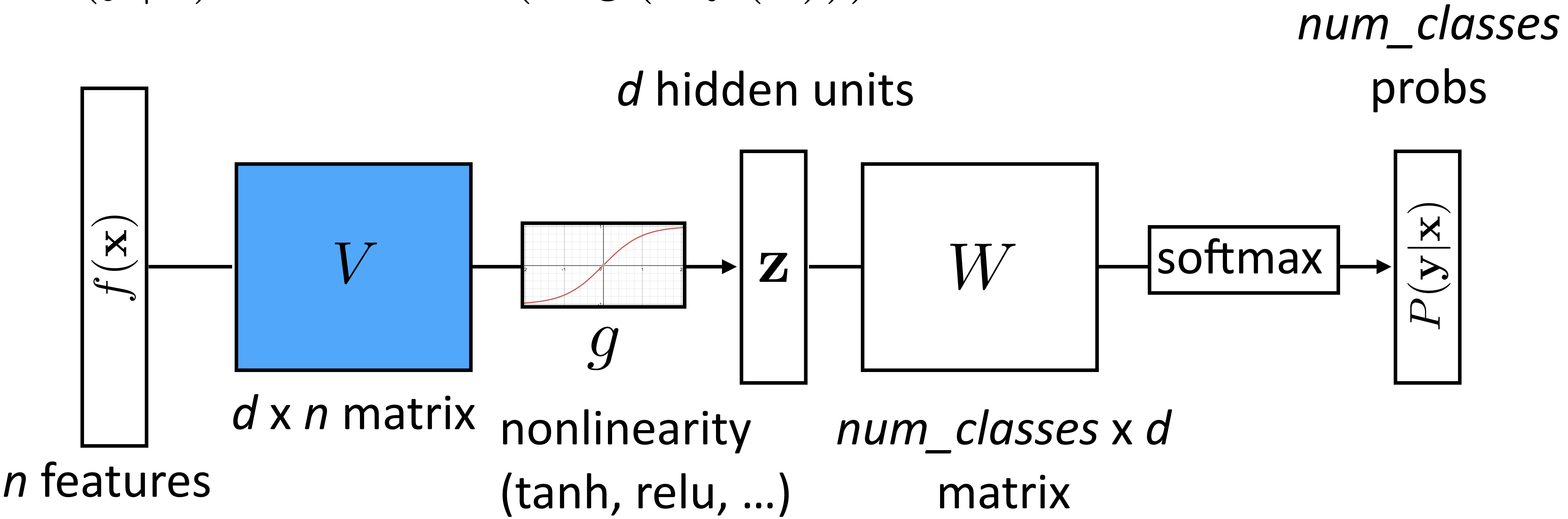


- Looks like logistic regression with ***z*** as the features!

$$dL(w) = [y - \sigma(w \cdot \phi(x)]\phi(x)$$

Neural Networks for Classification

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$$



- How should we compute gradient for intermediate parameters?

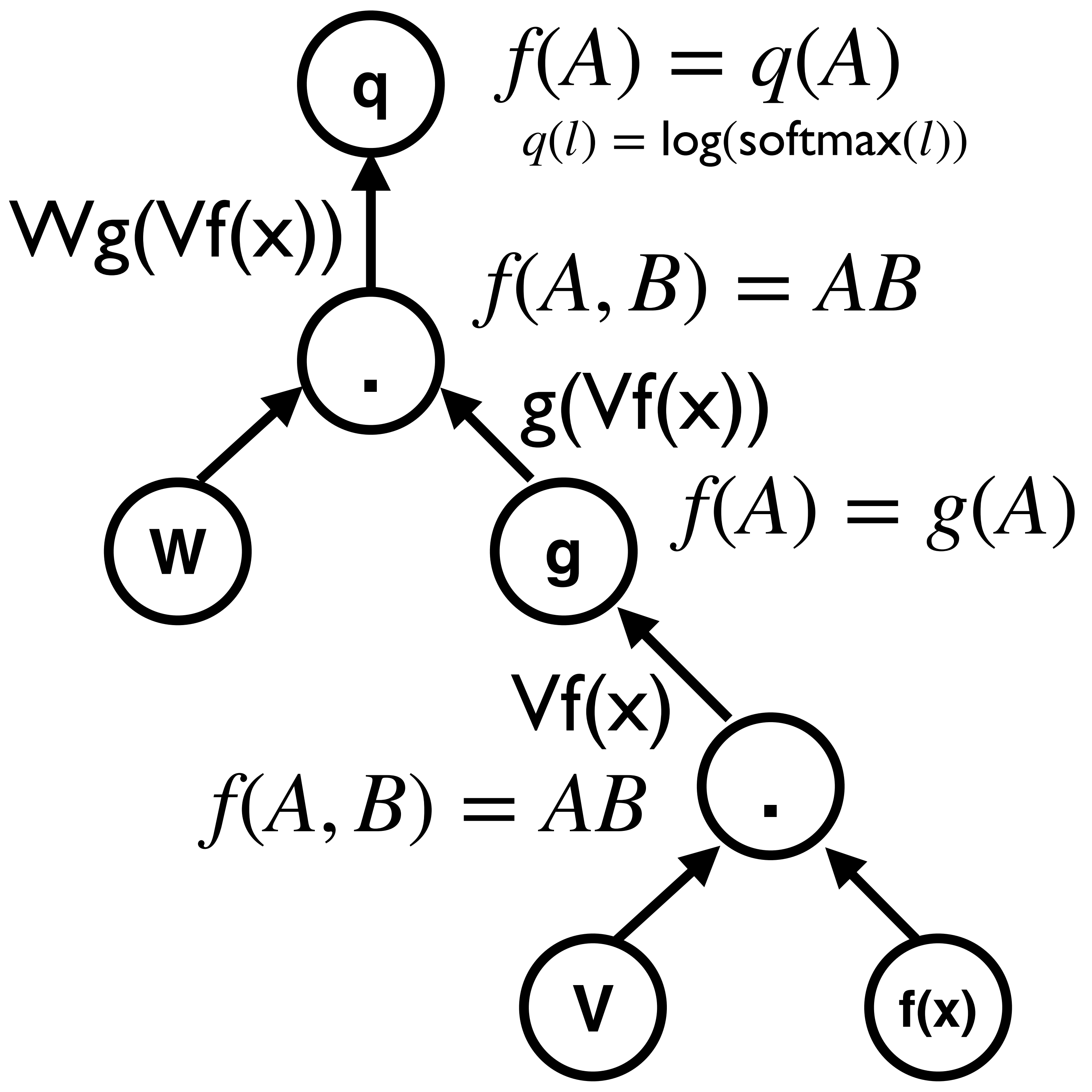
Today

- Introduction to neural network
- Introduction to computational graph
- Introduction to backpropagation
- Few practical tips..
  - training neural network
  - PyTorch introduction

Computational Graphs

- Functional description of the required computation for deep learning models

$$\log P(\mathbf{y}|\mathbf{x}) = \log(\,\mathrm{softmax}(W g(V f(\mathbf{x}))))$$



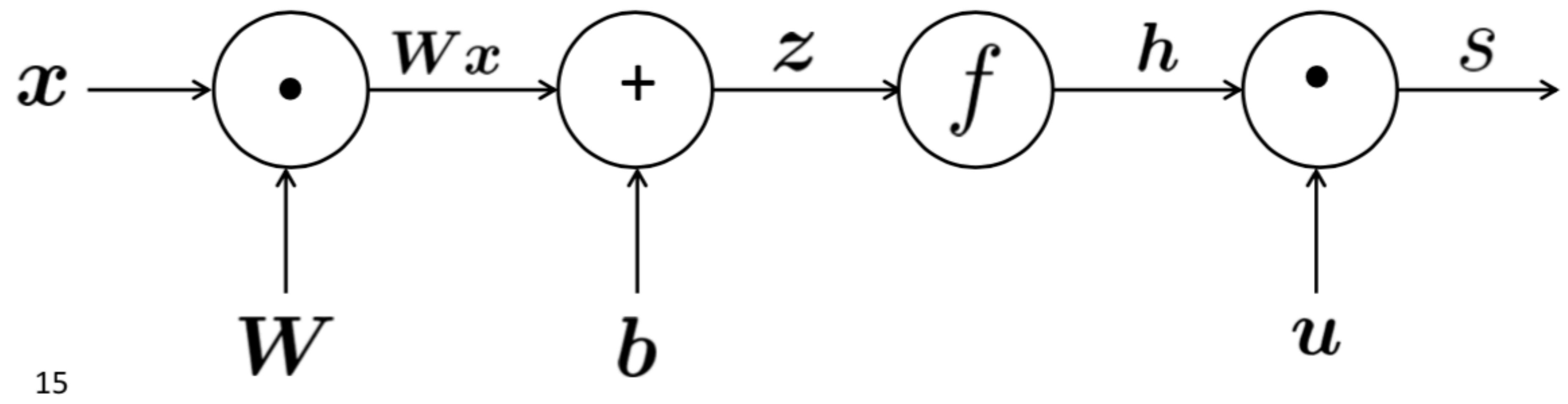
A **node** with an incoming **edge** is a **function** of that edge's tail node.

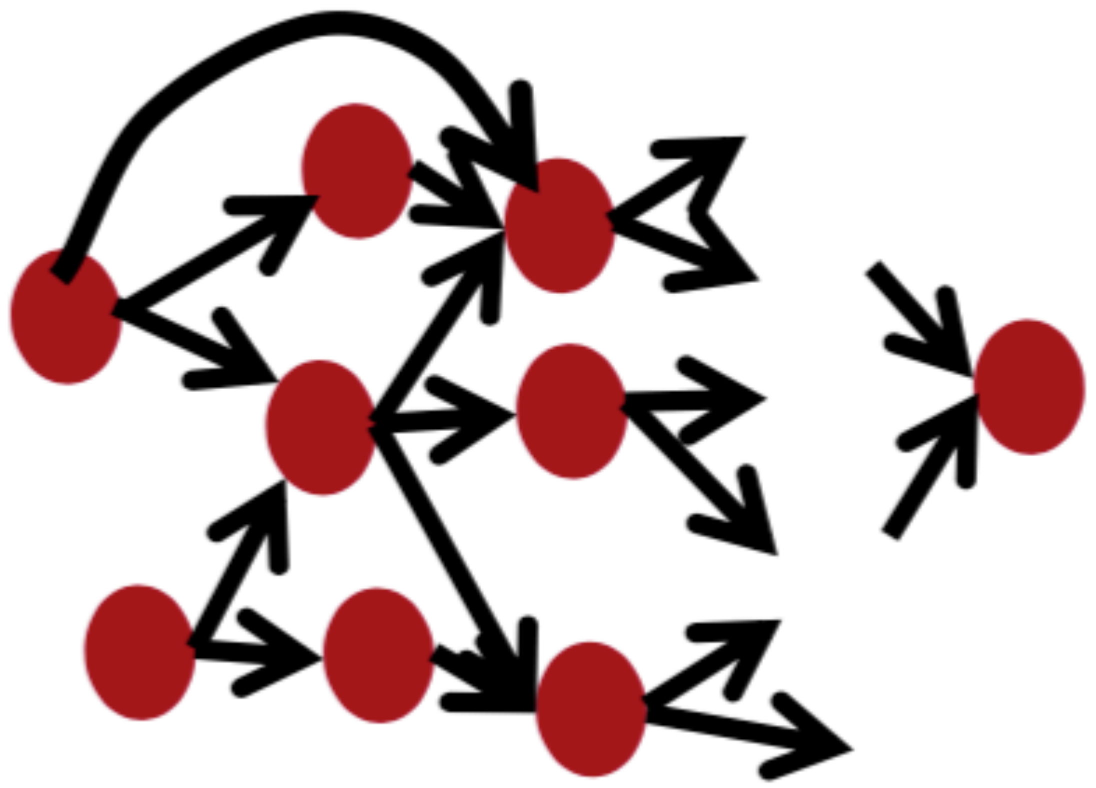
An **edge** represents a function argument (and also data dependency). They are just pointers to nodes.

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

Computational Graphs

Forward computation

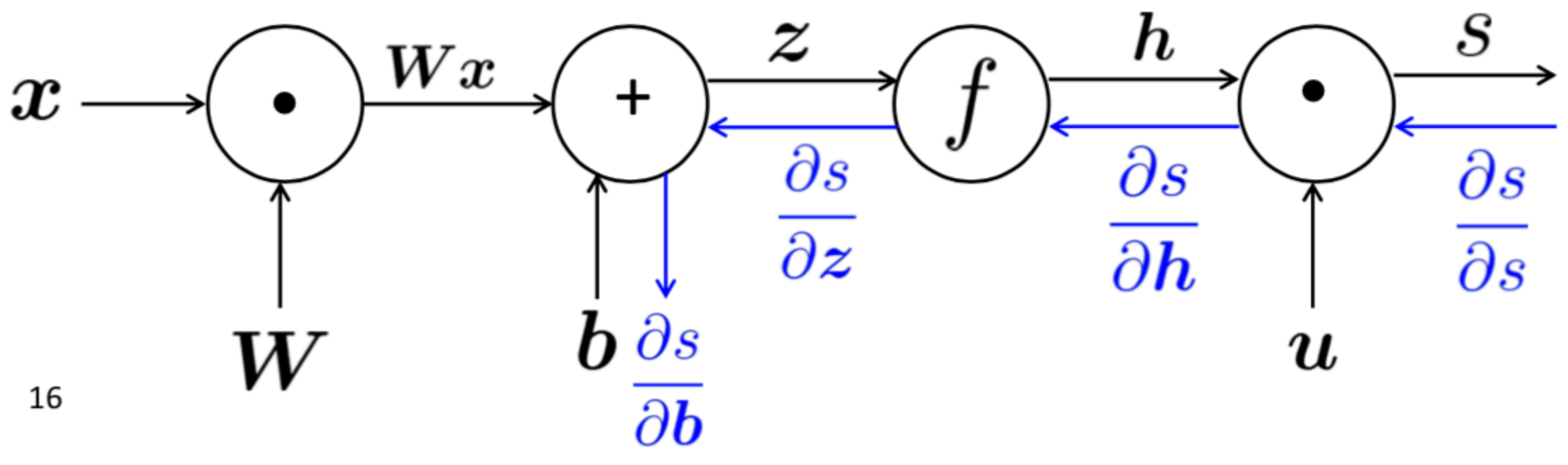


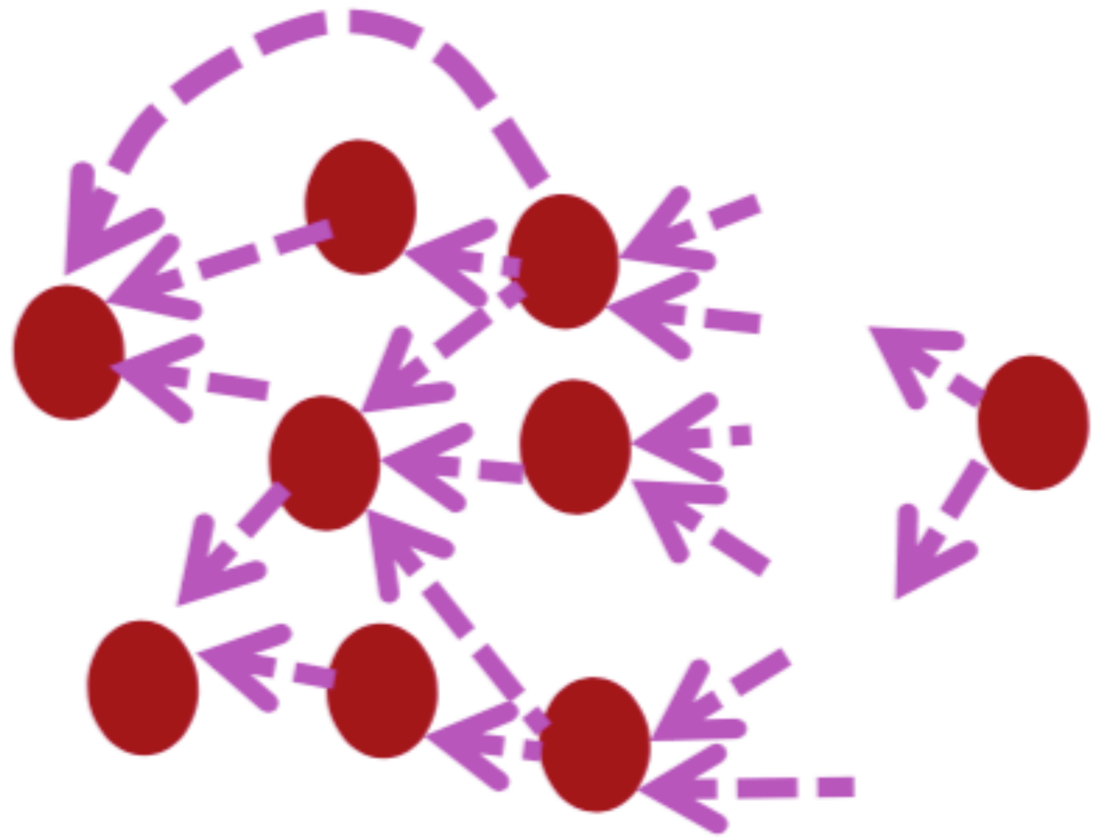


- Given parameters and input, make a prediction
- Visits nodes in topological order

Computational Graphs

Backward computation

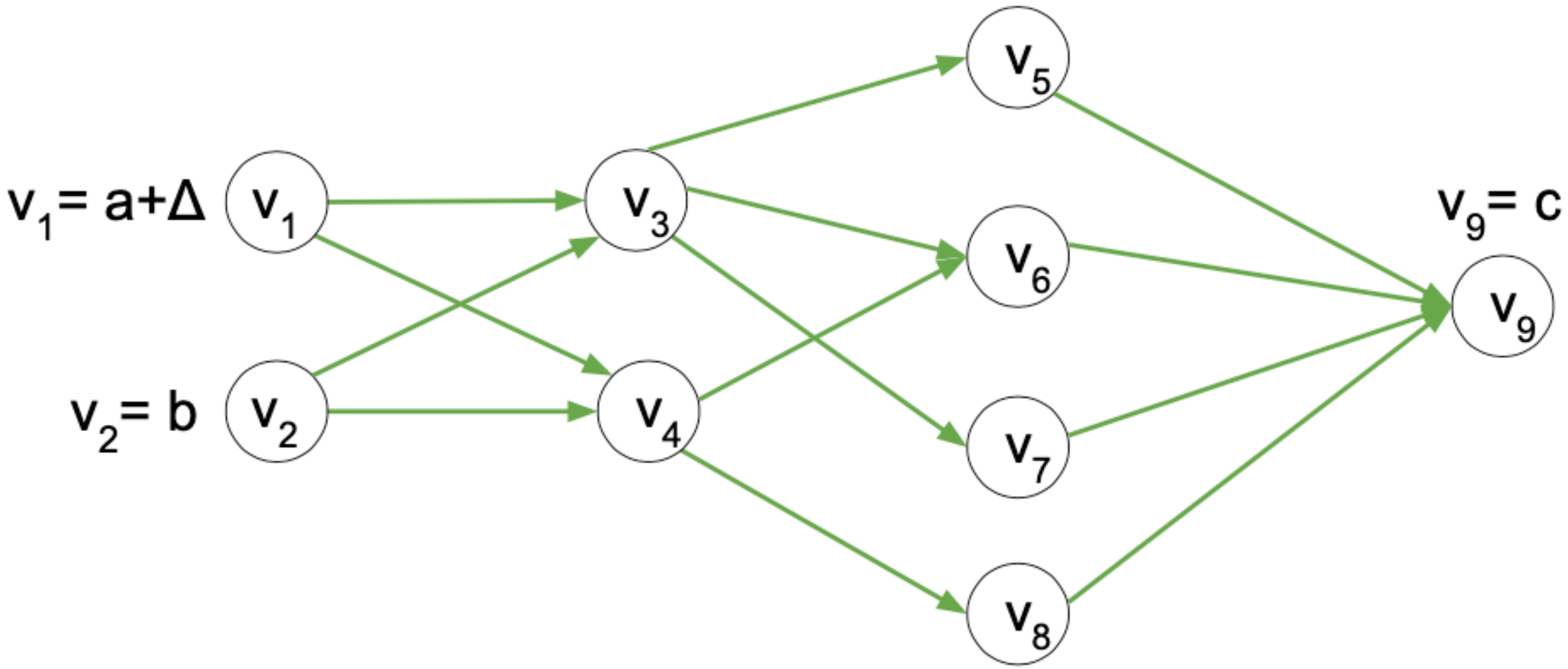




- Loop over the nodes in ***reverse*** topological order, starting from a final goal node (often our loss function)
- How does the output change if I make changes to the input?

Computing gradient on computational graph

- How many paths are there from v_1 to v_9?



This and a few followup slides credit on backpropagaion: Ryan Cotterell

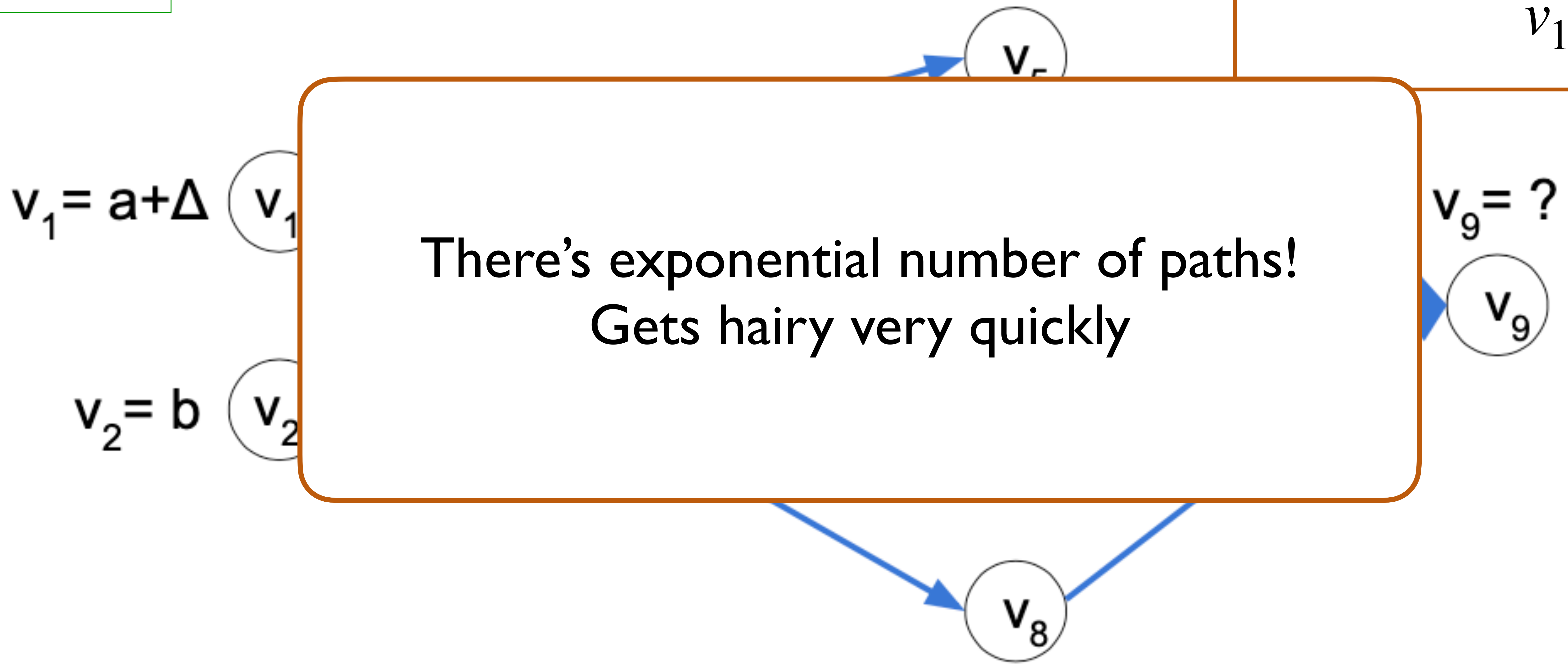
Computing gradient on computational graph

Chain rule

$$\frac{\partial}{\partial x} f(g(x)) = f'(g(x))g'(x)$$

$$\frac{\partial v_9}{\partial v_1} = \sum_{v_1, v_i, \ldots, v_j, v_9} \frac{\partial v_i}{\partial v_1} \cdots \frac{\partial v_9}{\partial v_j}$$

Sum over all paths in the computation graph from v_1 to v_9



There's exponential number of paths!
Gets hairy very quickly

Solution

- Dynamic programming!
- Instead of considering exponentially many paths between the weight and final loss, store and reuse the intermediate results
- Starts from the end of the computational graph
- Visit nodes in reverse topological order and compute gradient w.r.t each node using gradient w.r.t successors

$$\frac{\partial L}{\partial x} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial y_i} \frac{\partial y_i}{\partial x}$$

$$\{y_1, \ldots, y_n\} = \text{ successors of } x$$

Backpropagation

- Key idea 1: Use the chain rule

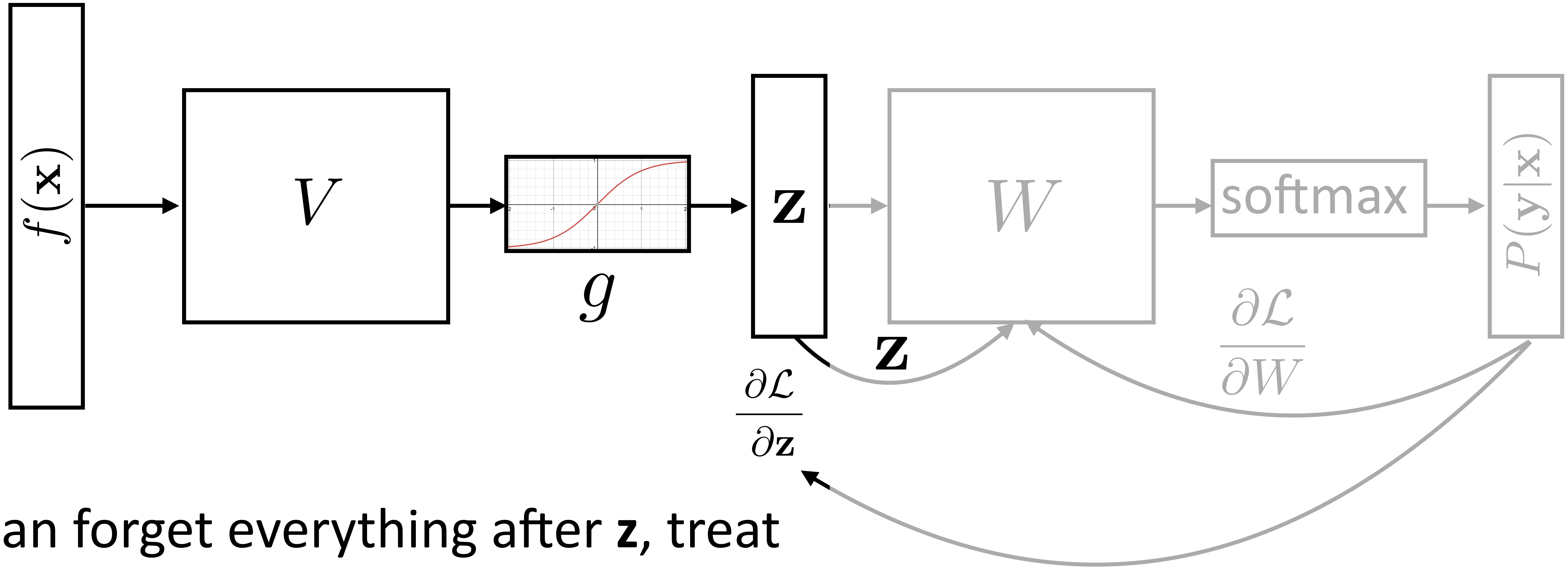
$$\frac{\partial}{\partial x} f(g(x)) = f'(g(x))g'(x)$$

- Key Idea 2: **Re-using** derivatives computed from the later layers in computing derivations for lower layers, allowing efficient computation of gradients

https://people.cs.umass.edu/~domke/courses/sml2011/08autodiff_nnets.pdf

Example

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



- Can forget everything after **z**, treat it as the output and keep computing gradients

Example: Backpropagation

$$\mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W\mathbf{z}) \cdot e_j$$

$\mathbf{z} = g(Vf(\mathbf{x}))$

Activations at hidden layer

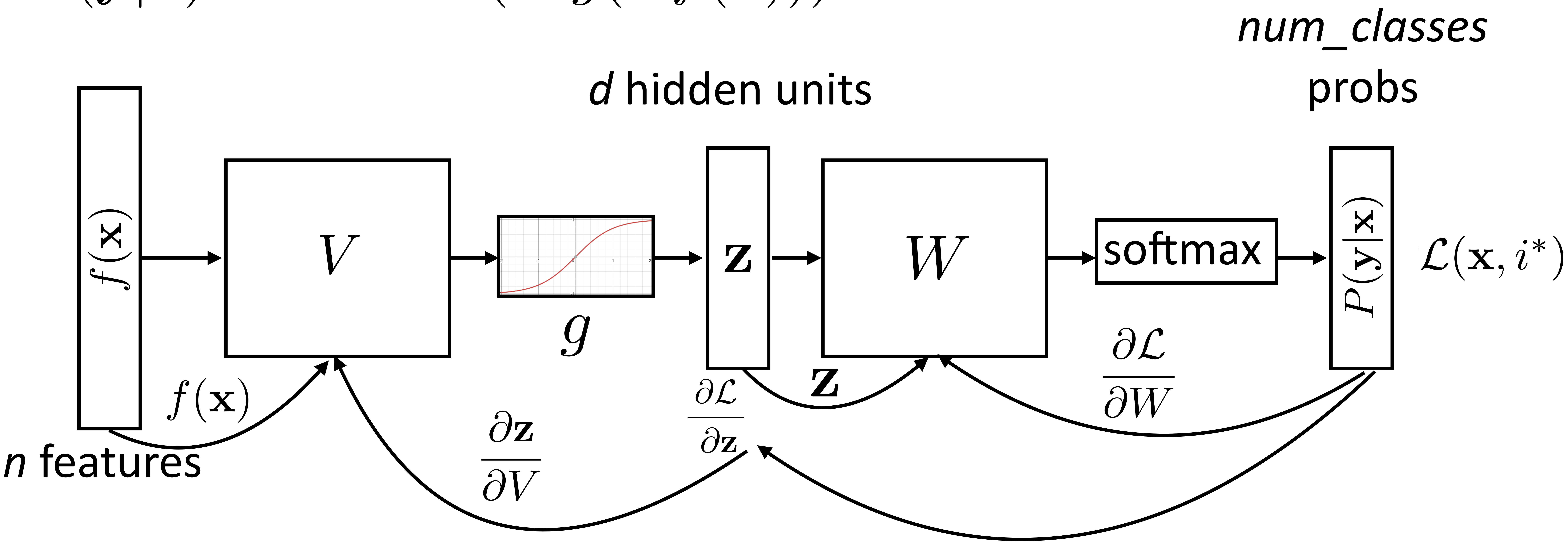
- Gradient with respect to *V*: apply the chain rule

$$\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial V_{ij}} = \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial V_{ij}} \qquad \frac{\partial \mathbf{z}}{\partial V_{ij}} = \frac{\partial g(\mathbf{a})}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial V_{ij}} \qquad \mathbf{a} = Vf(\mathbf{x})$$

- First term: gradient of nonlinear activation function at ***a*** (depends on current value)
- Second term: gradient of linear function
- First term: represents gradient w.r.t. **z** $\quad \frac{\partial \mathcal{L}}{\partial \mathbf{z}} \quad W^\top(y - y^*)$

Backpropagation: Picture

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



$$\frac{\partial \mathcal{L}}{\partial V} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial V}$$

Backpropagation: History

- Building blocks dates back to:
  - Chain Rule (1676, Leibniz)
  - Dynamic Programming (DP, Bellman, 1957)
  - Gradient Descent (Cauchy 1847,…)
- Explicit, efficient error propagation for neural network (1970, 1982)
- Rumelhart, Hinton and William 1986, LeCun 1985
  - Backpropagation for neural network becomes popular (as computes improved, By 1985, compute was about 1,000 times cheaper than in 1970!

https://people.idsia.ch//~juergen/who-invented-backpropagation.html

Following few slides adapted from Ryan Cotterell

Today

- Introduction to neural network
- Introduction to computational graph
- Introduction to backpropagation
- Few practical tips..
  - training neural network
  - PyTorch introduction

Training Neural Network

- The learning object is no longer convex once we introduce non-linearity functions.

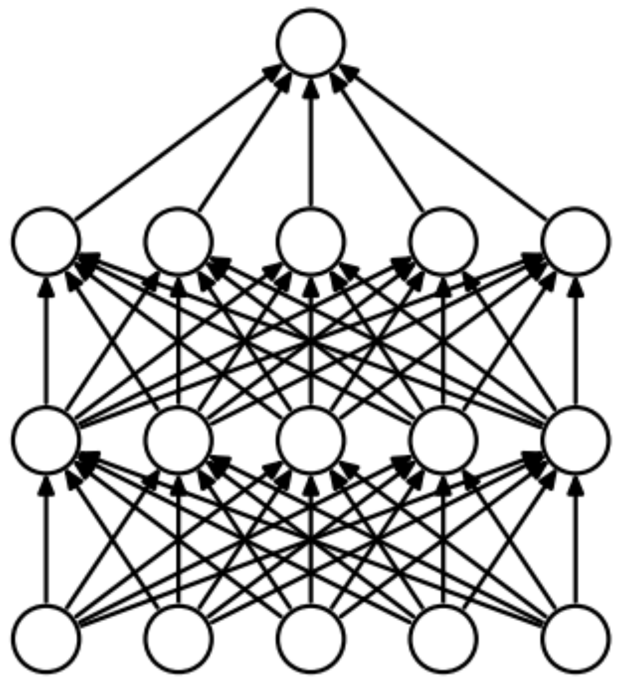
- Basic formula: compute gradients on batch, use optimization method (Stochastic Gradient Descent, Adagrad, etc.)

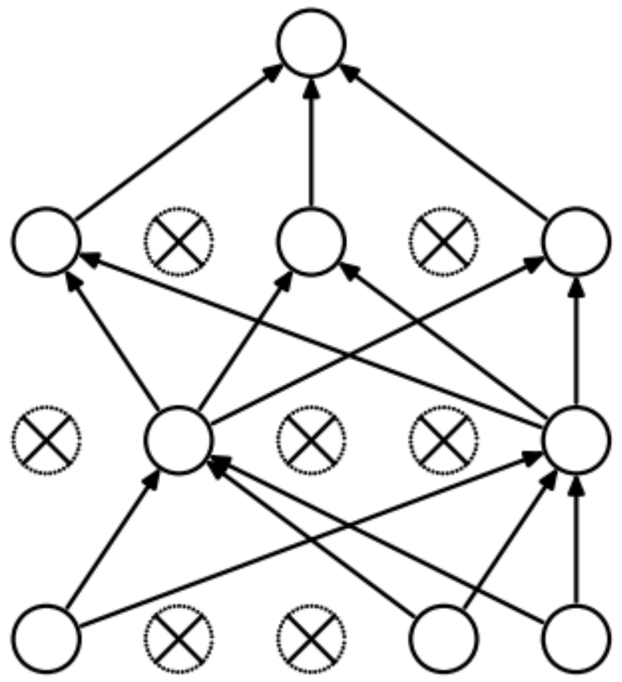
- Questions:
  - How to initialize? How to regularize? What optimizer to use?

- Few practical tips today, take deep learning or optimization courses to understand this further

Learning Tricks

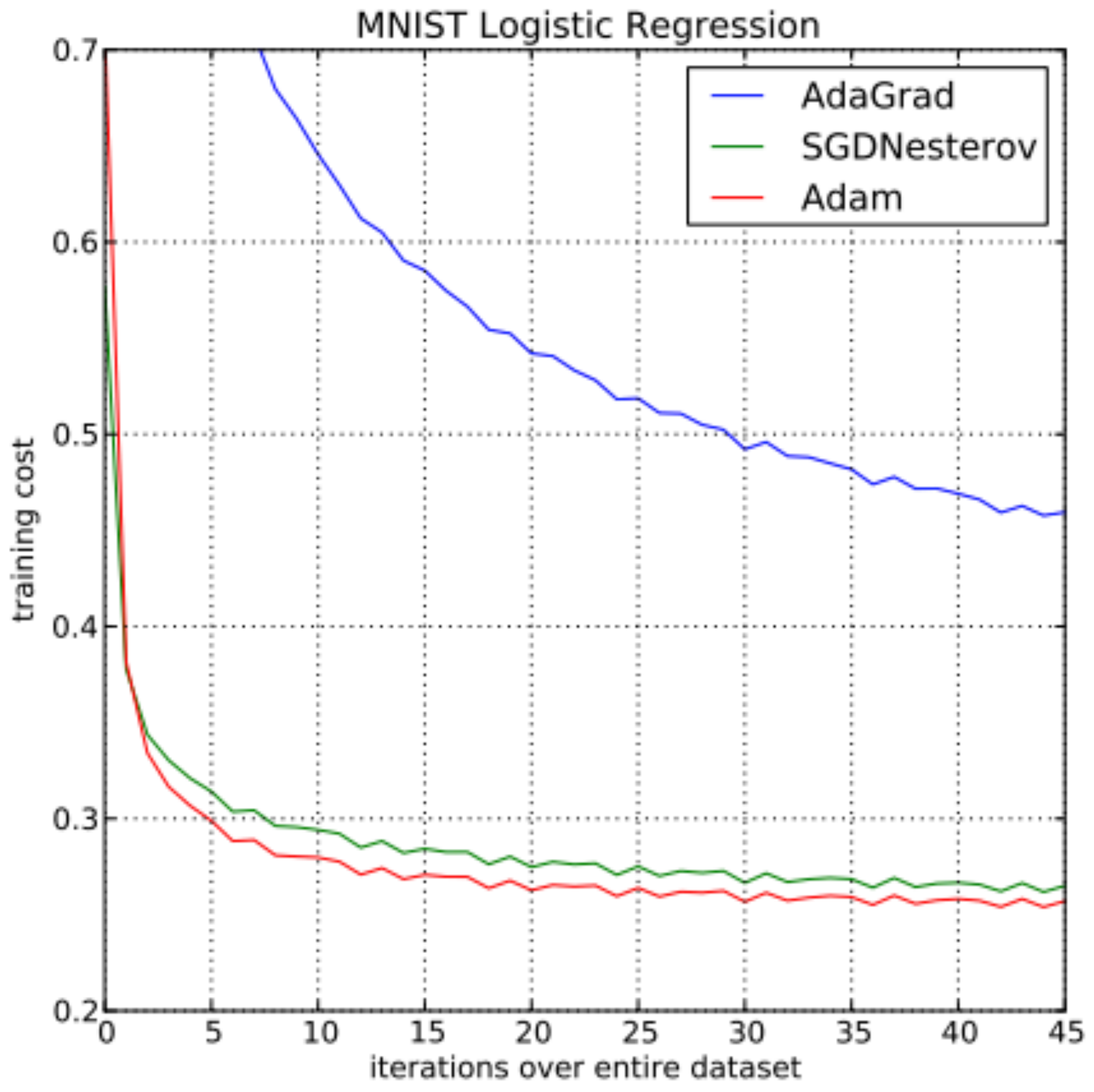
- Initialization:
  - Initialize too large and cells are saturated, uninformative gradients
  - Random uniform / normal initialization with appropriate scale
  - Fancier initialization (e.g., Xavier initialization) can help
- Normalizaton:
  - Want variance of inputs and gradients for each layer to be the same
  - Different techniques (e.g., Batch normalization, layer normalization, etc)
- Regularizaton:
  - Dropout: Probabilistically zero out parts of the network during training to prevent overfitting, use whole network at test time

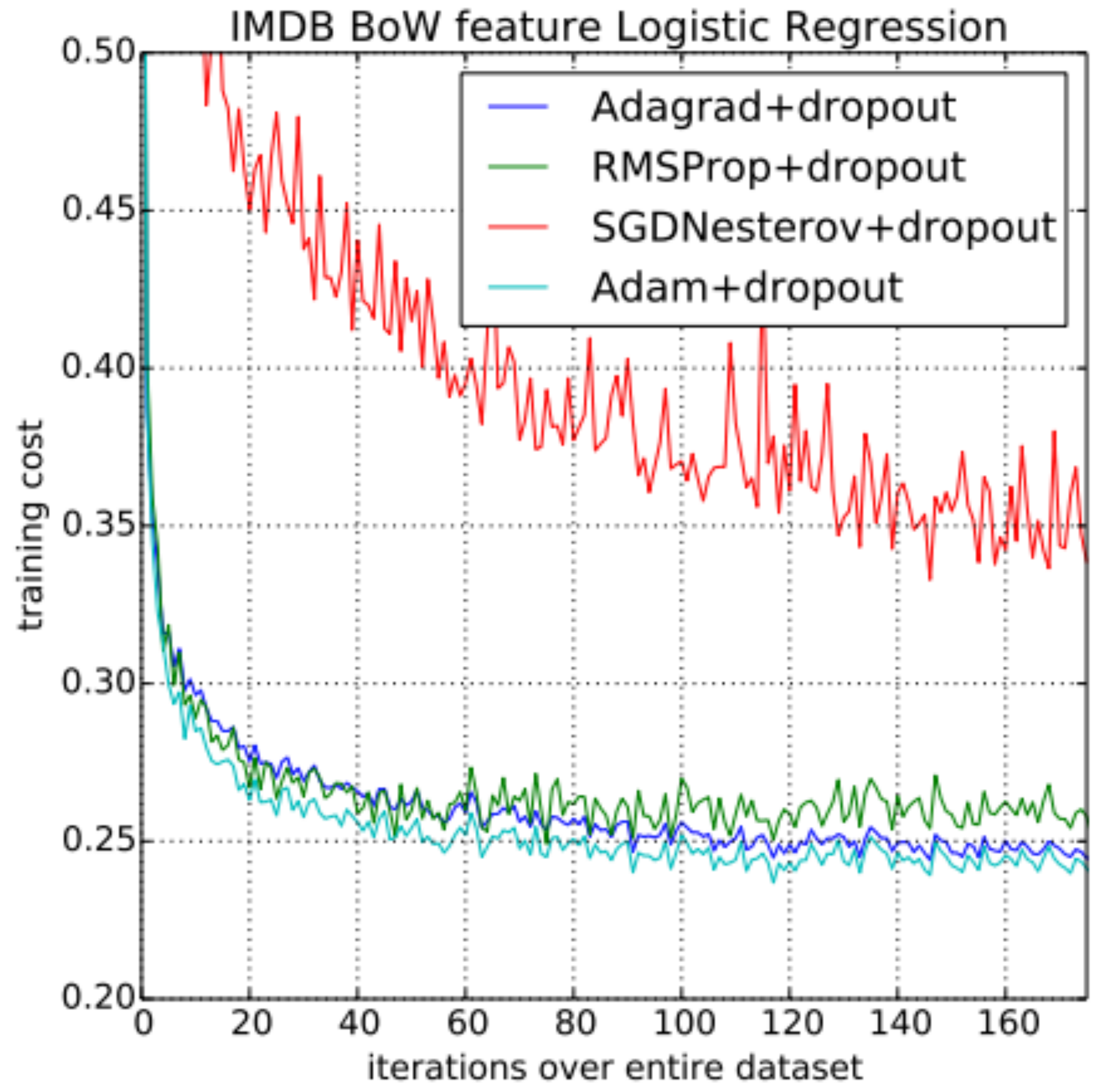

(a) Standard Neural Net


(b) After applying dropout.

Learning Tricks

- A class of more sophisticated “adaptive” optimizers that scale the parameter adjustment by an accumulated gradient.
  - Adam
  - Adagrad
  - Adadelta
  - RMSprop





- One more trick: **gradient clipping** (set a max value for your gradients)

PyTorch

- Framework for defining computations that provides easy access to derivatives
- Module: defines a neural network

```
torch.nn.Module

    # Takes an example x and computes result
    forward(x):
        ...

    # Computes gradient after forward() is called
    backward(): # produced automatically
        ...
```

Computation Graphs in Pytorch

- Define forward pass for $P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$

```
class FFNN(nn.Module):
    def __init__(self, input_size, hidden_size, out_size):
        super(FFNN, self).__init__()
        self.V = nn.Linear(input_size, hidden_size)
        self.g = nn.Tanh() # or nn.ReLU(), sigmoid()...
        self.W = nn.Linear(hidden_size, out_size)
        self.softmax = nn.Softmax(dim=0)
        nn.init.uniform(self.V.weight)

    def forward(self, x):
        return self.softmax(self.W(self.g(self.V(x))))
```

Input to Network

- Whatever you define with torch.nn needs its input as some sort of tensor, whether it's integer word indices or real-valued vectors

```
def form_input(x) -> torch.Tensor:
  # Index words/embed words/etc.
  return torch.from_numpy(x).float()
```

- torch.Tensor is a different data structure from a numpy array, but you can translate back and forth fairly easily
- Note that **translating out of PyTorch will break backpropagation**; don't do this inside your Module

Training and Optimization

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$$

one-hot vector of the label (e.g., [0, 1, 0])

```
ffnn = FFNN(inp, hid, out)
optimizer = optim.Adam(ffnn.parameters(), lr=lr)
for epoch in range(0, num_epochs):
    for (input, gold_label) in training_data:
        ffnn.zero_grad() # clear gradient variables
        probs = ffnn.forward(input)
        loss = torch.neg(torch.log(probs)).dot(gold_label)
        loss.backward()
        optimizer.step()
```

negative log-likelihood of correct answer

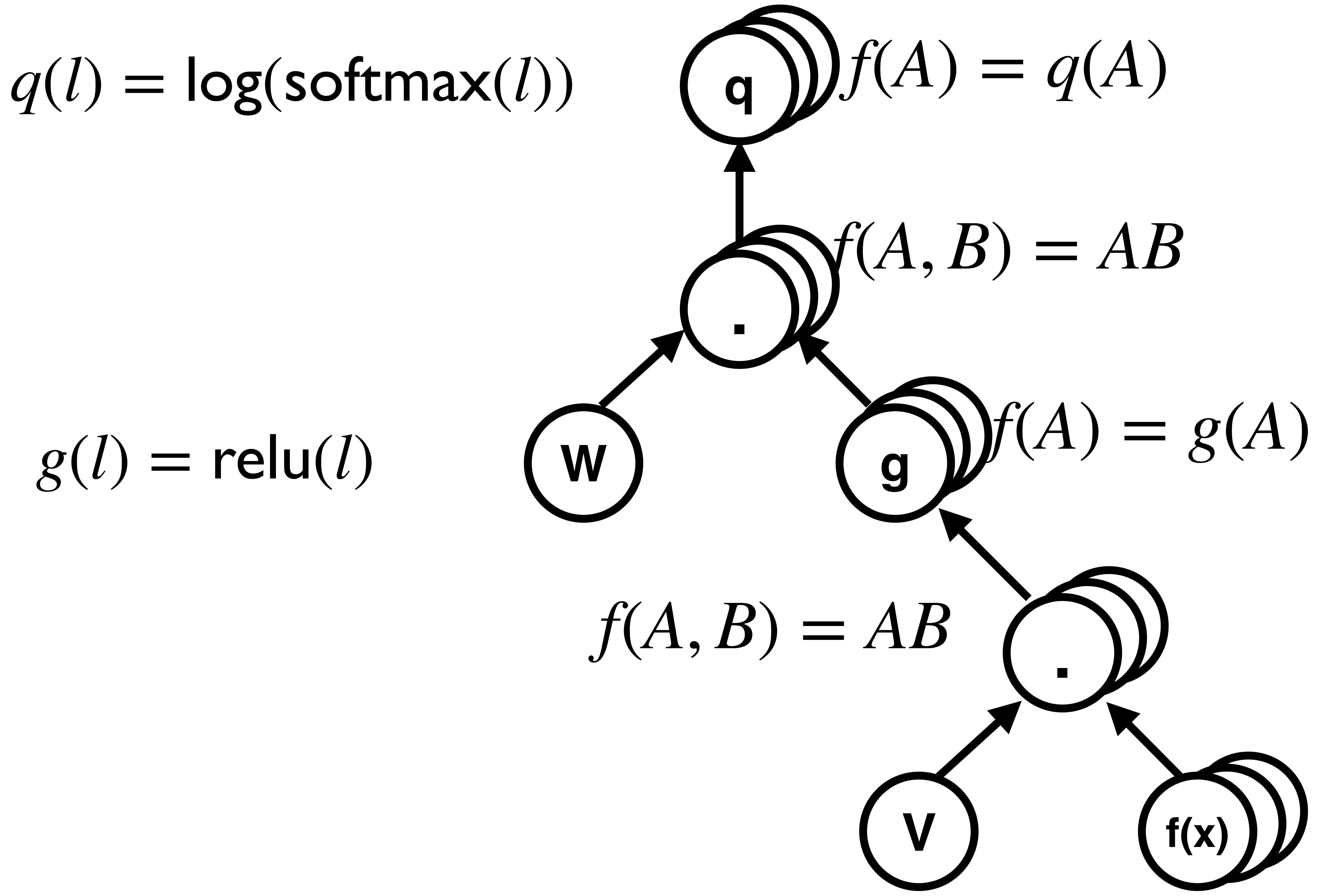
Batching

- Packing multiple examples together to have computational benefits
- A lot more meaningful in GPU (using all the GPU cores!)
- Batching becomes a bit tricker if the network becomes complex
- Batching should not add new dimensions to the parameters!

```
# input is [batch_size, num_feats]
# gold_label is [batch_size, num_classes]
def make_update(input, gold_label)
    ...
    probs = ffnn.forward(input) # [batch_size, num_classes]
    loss = torch.sum(torch.neg(torch.log(probs)).dot(gold_label))
    ...
```

Visualizing Batching in computational graph

$$\log P(\mathbf{y}|\mathbf{x}) = \log(\mathrm{softmax}(W g(V f(\mathbf{x}))))$$



Training a Model

Define computational graph

Initialize weights and optimizer

For each epoch:

  For each batch of data:

    Zero out gradient

    Compute loss on batch

    Autograd to compute gradients and take step on optimizer

  [Optional: check performance on dev set to identify overfitting]

Run on dev/test set