Back-propagation and Computation Graph

Yoav Goldberg

Reminder: gradient based training

- Computing the gradients:
  - The network (and loss calculation) is a mathematical function.

$$\ell(x,k) = -log(softmax(\mathbf{W}^3 g^2(\mathbf{W}^2 g^1(\mathbf{W}^1 x + \mathbf{b}^1)+\mathbf{b}^2) + \mathbf{b}^3)[k])$$

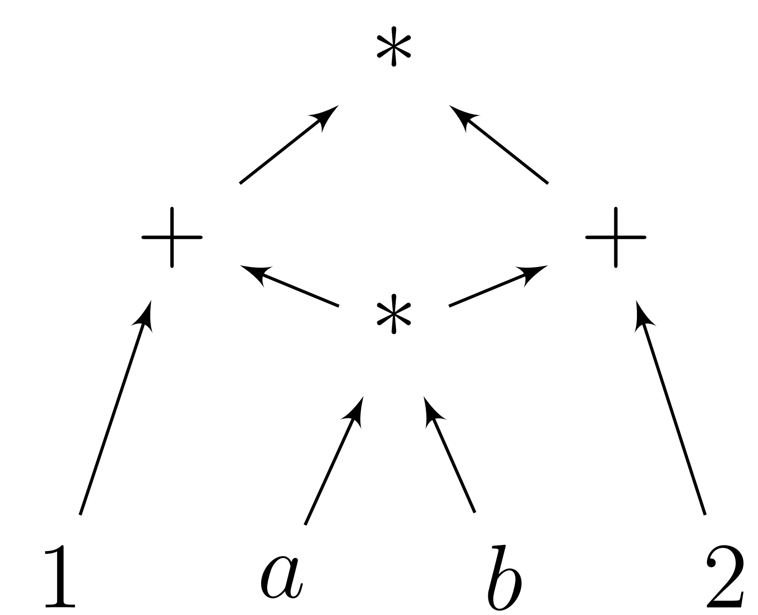
  - Calculus rules apply.
  - (a bit hairy, but carefully follow the chain rule and you'll get there)

(chain rule - on whiteboard)

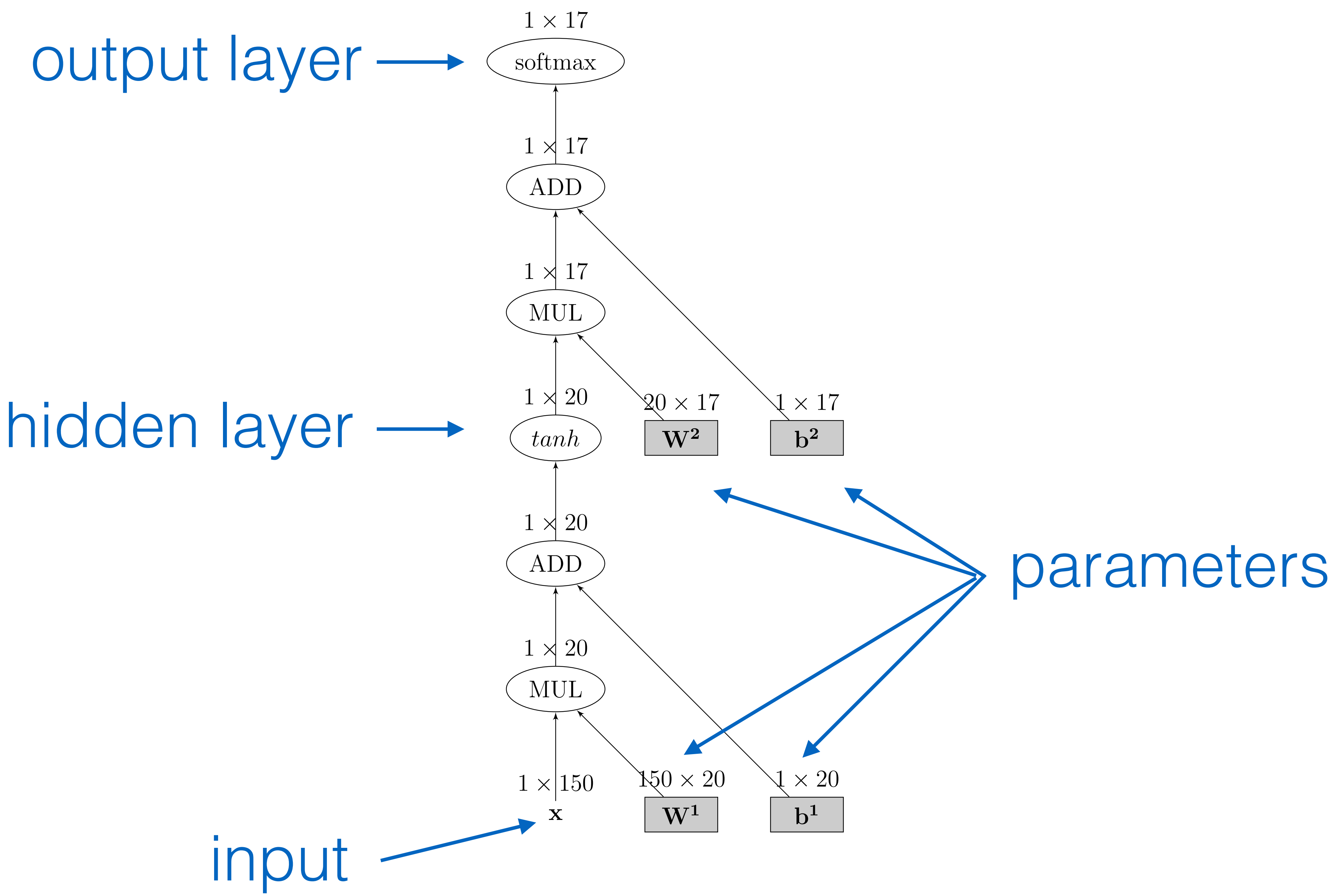
The Computation Graph (CG)

- a DAG.
- Leafs are inputs (or parameters).
- Nodes are operators (functions).
- Edges are results (values).
- Can be built for any function.

$$(a * b + 1) * (a * b + 2)$$

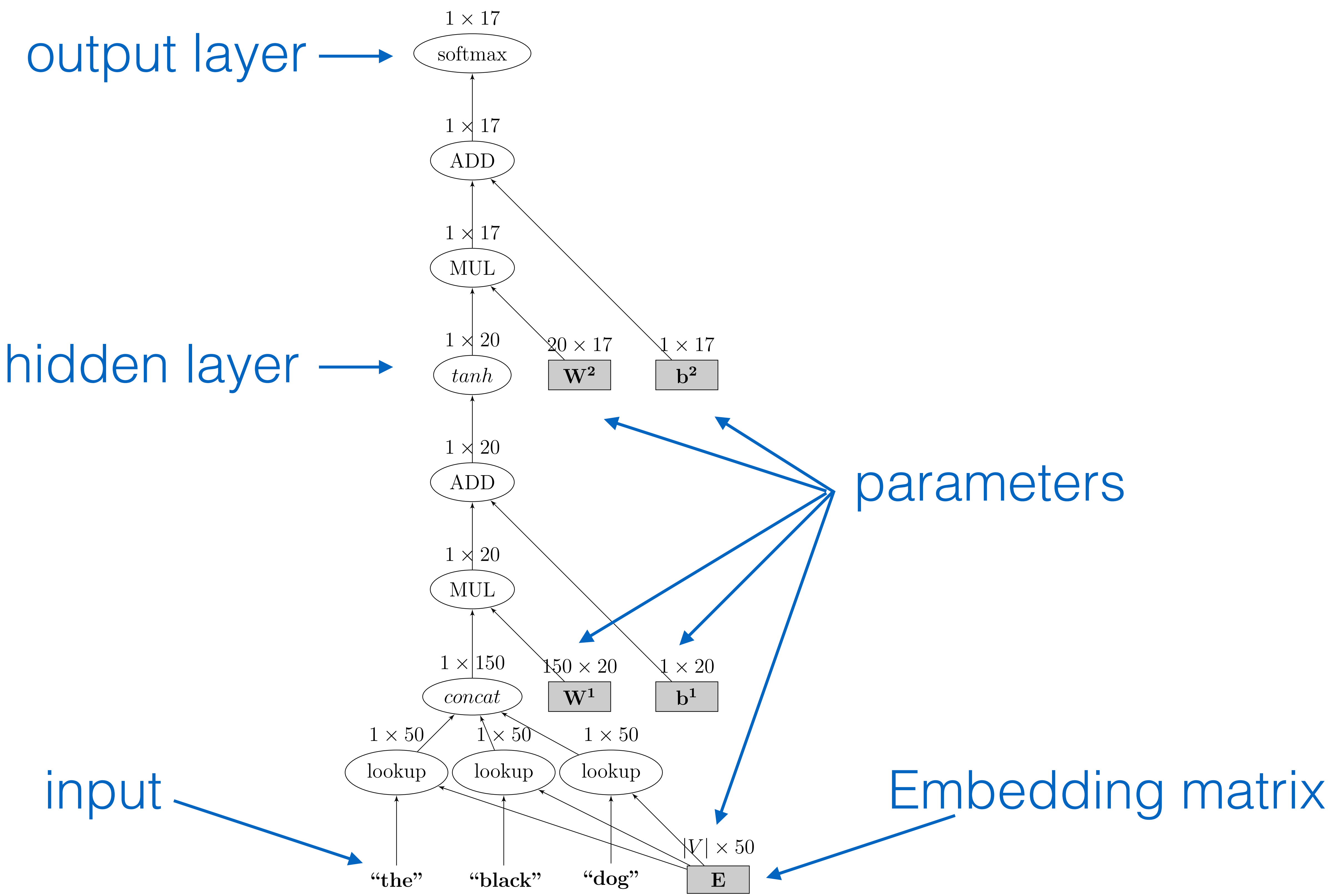


MLP_1

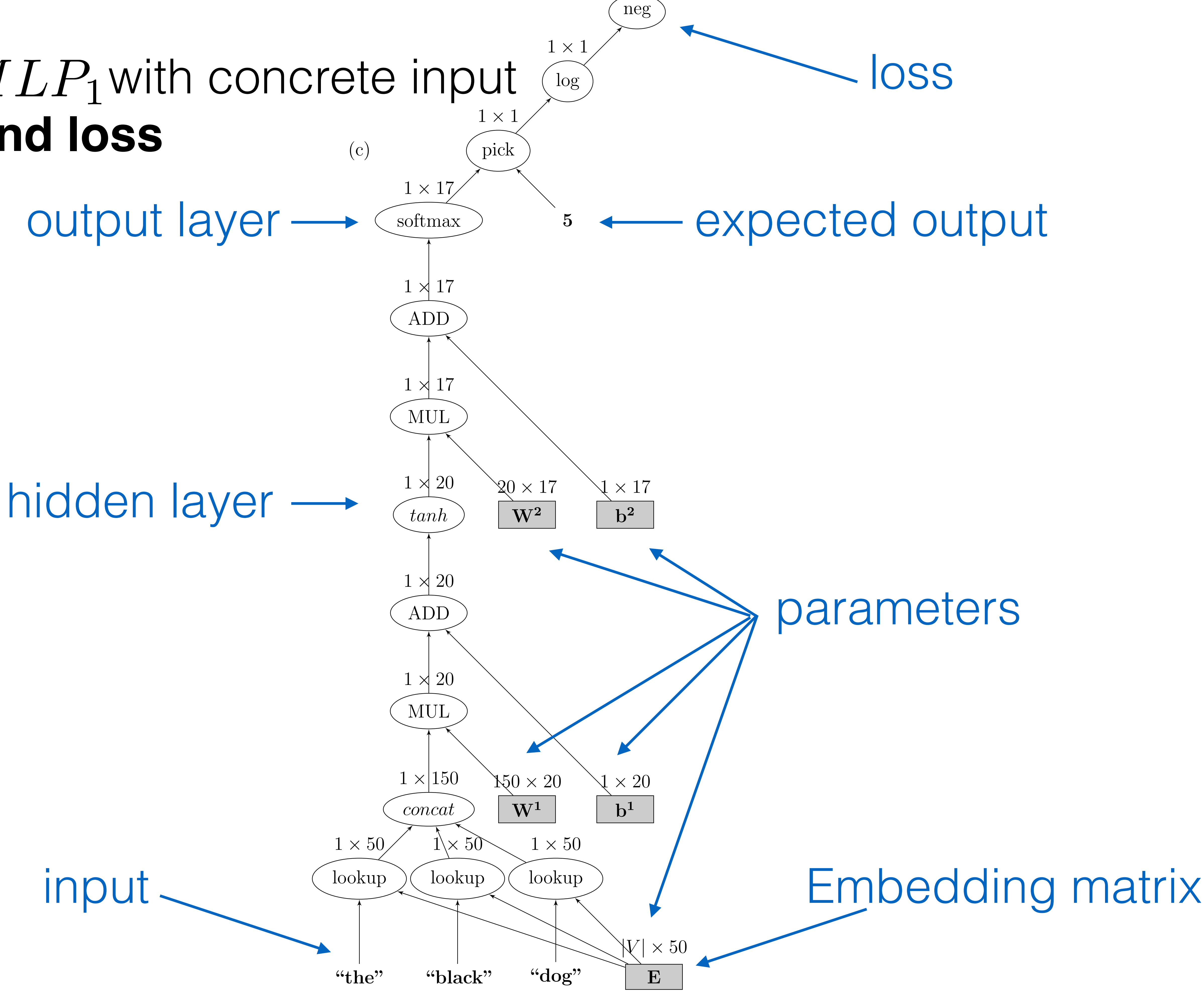


output layer
1 × 17
softmax
1 × 17
ADD
1 × 17
MUL
hidden layer
1 × 20
tanh
20 × 17
W²
1 × 17
b²
parameters
1 × 20
ADD
1 × 20
MUL
1 × 150
x
150 × 20
W¹
1 × 20
b¹
input

MLP_1 with concrete input



MLP_1 with concrete input **and loss**

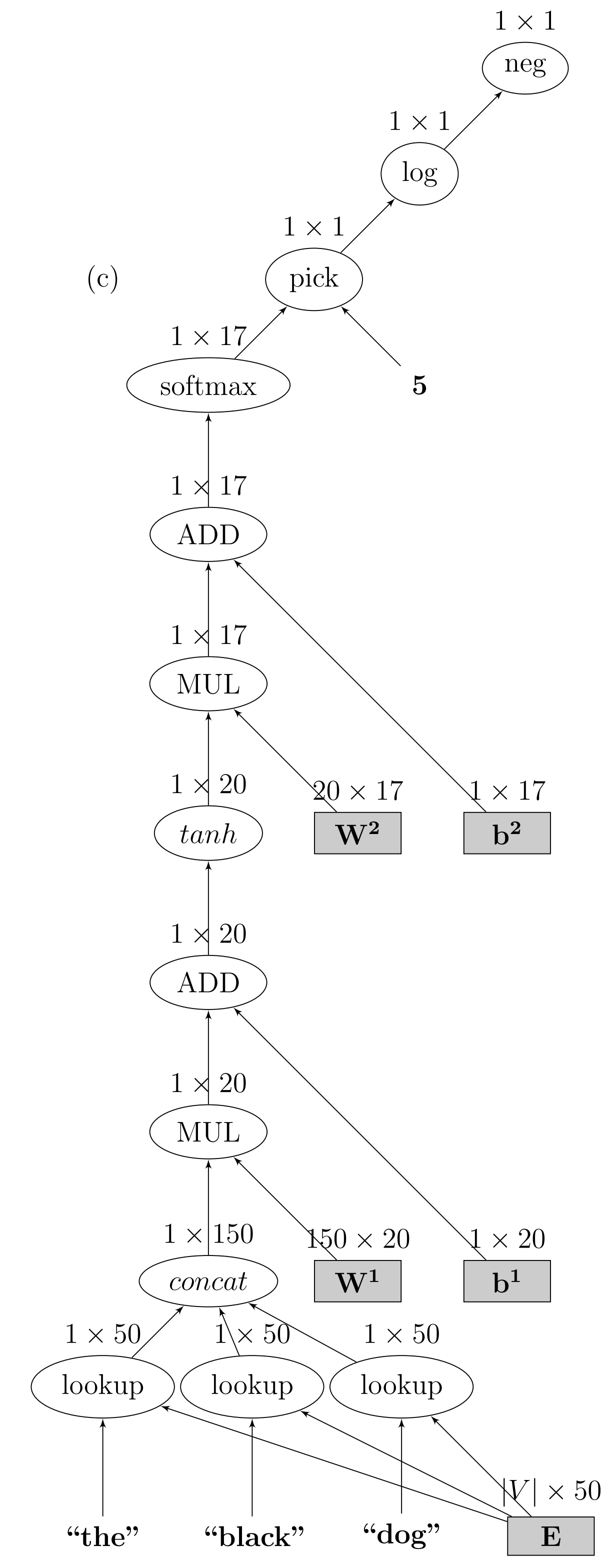


- Create a graph for each training example.

- Once graph is built, we have two essential algorithms:

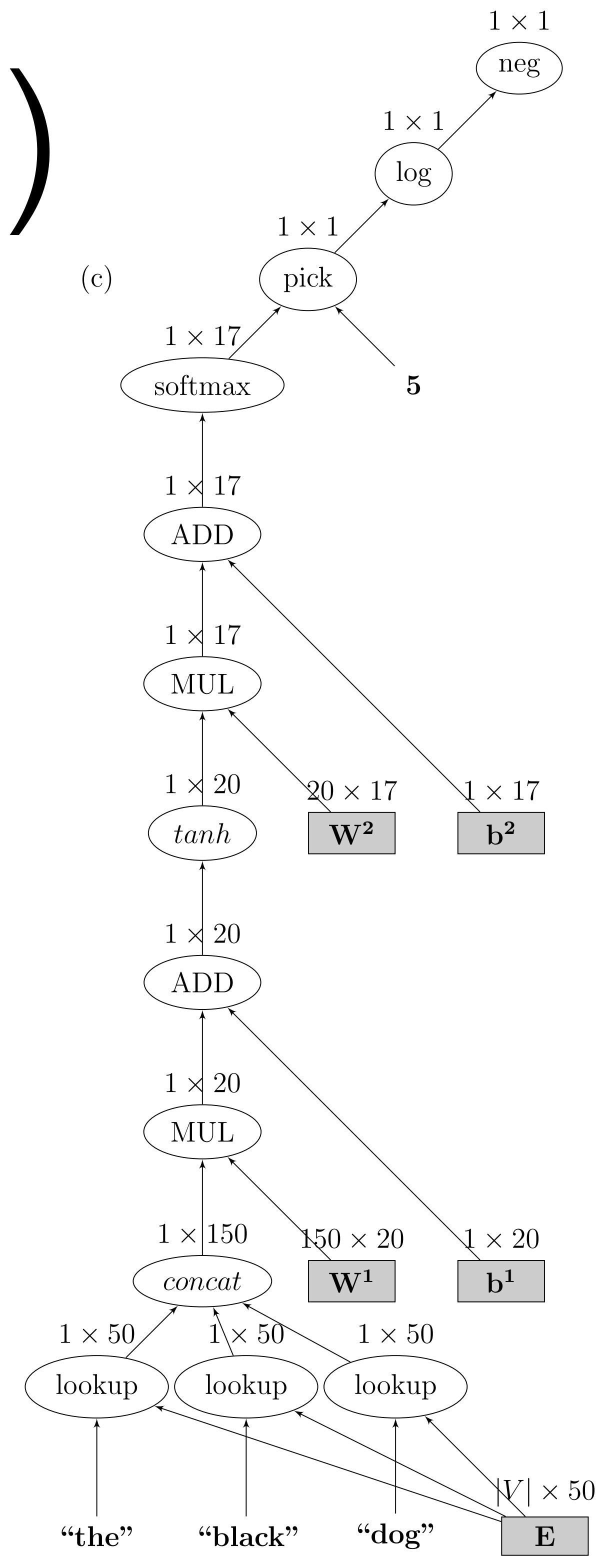
  - **Forward:** compute all values.

  - **Backward (backprop):** compute all gradients.



Computing the Gradients (backprop)

- Consider the chain-rule (example on blackboard)
- Each node needs to know how to:
    - Compute forward.
    - Compute its **local** gradient.



CG Software Packages

- Theano (Bengio's lab, python, low level, grandfather of CG, retired).
- Torch (Lua, wide support, Facebook backed, very fast on GPU, almost retired)
- Tensor Flow (Google, python / C++ hybrid)
- Chainer (python)
- PyTorch (python, dynamic, by Facebook)
- DyNet (C++/Python, by Chris Dyer, Graham Neubig, and Yoav Goldberg)
- Keras (python, high level, theano/TF backends)

shines for dynamic graphs, recursive nets

best bet for out-of-the-box models

BIU NLP

The Python Neural Networks Toolkits Landscape (partial)

theano

TensorFlow

K

AllenNLP

Chainer

dy/net

PyTorch

The Python Neural Networks Toolkits Landscape (partial)

high-level

theano

TensorFlow

K

low-level

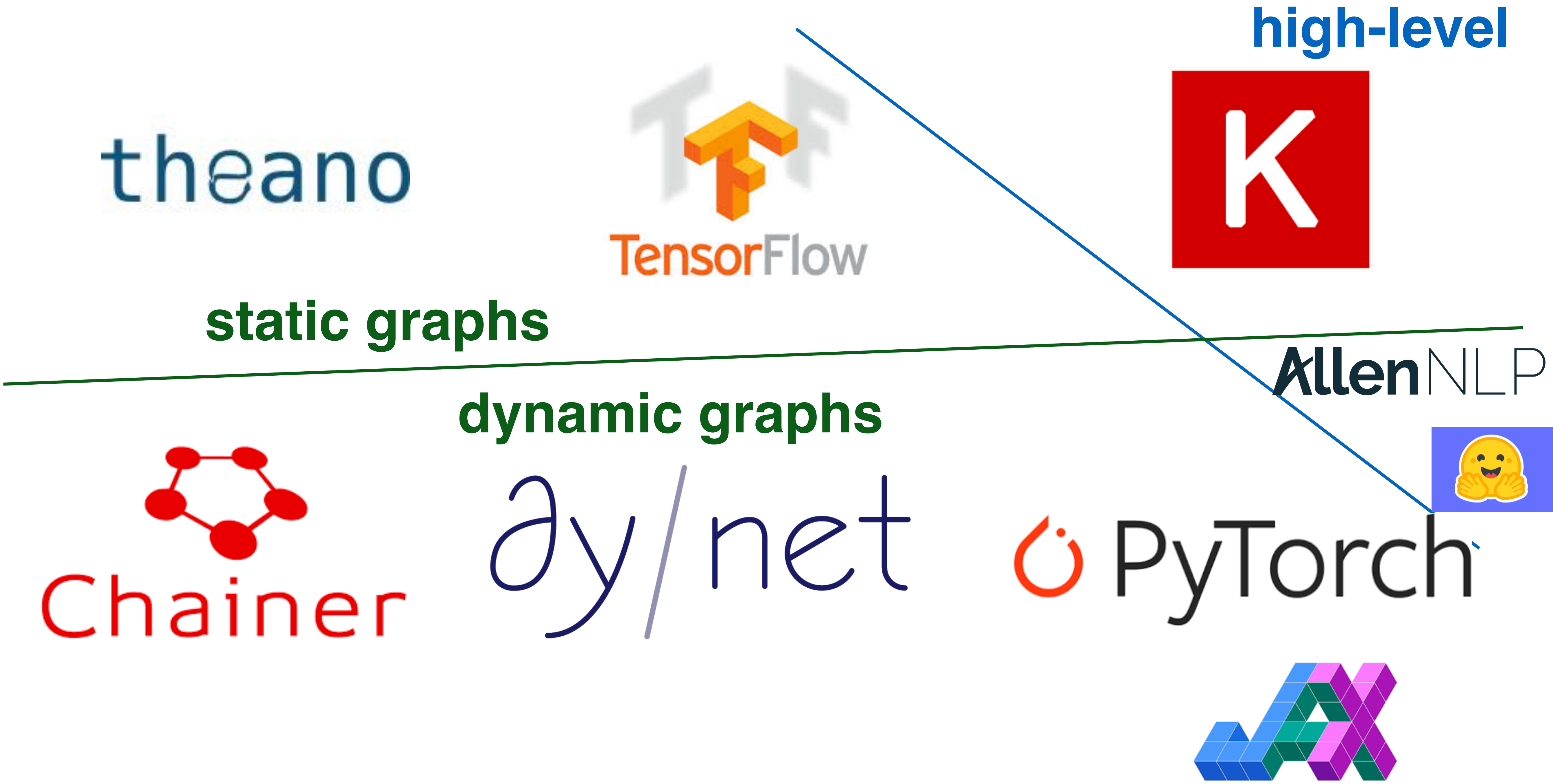
AllenNLP

Chainer

dy/net

PyTorch

The Python Neural Networks Toolkits Landscape (partial)



The Python Neural Networks Toolkits Landscape (partial)

high-level

static graphs

dynamic graphs

- fast also on CPU
- automatic batching

The Python Neural Networks Toolkits Landscape (partial)

high-level

theano

TensorFlow

K

static graphs

AllenNLP

dynamic graphs

Chainer

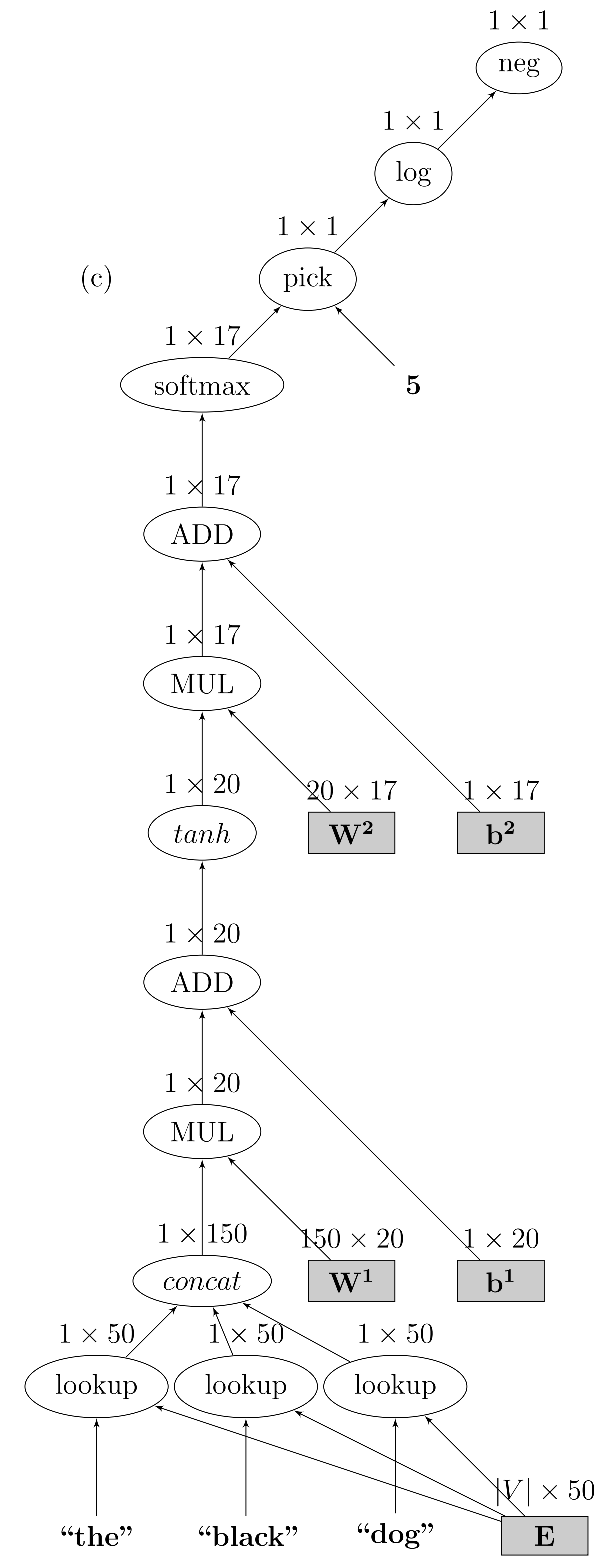
dy/net

PyTorch

- This is what the world is using today.

Network Training algorithm:

- For each training example (or mini-batch):
  - Create graph for computing loss.
  - Compute loss (**forward**).
  - Compute gradients (**backwards**).
  - Update model parameters.



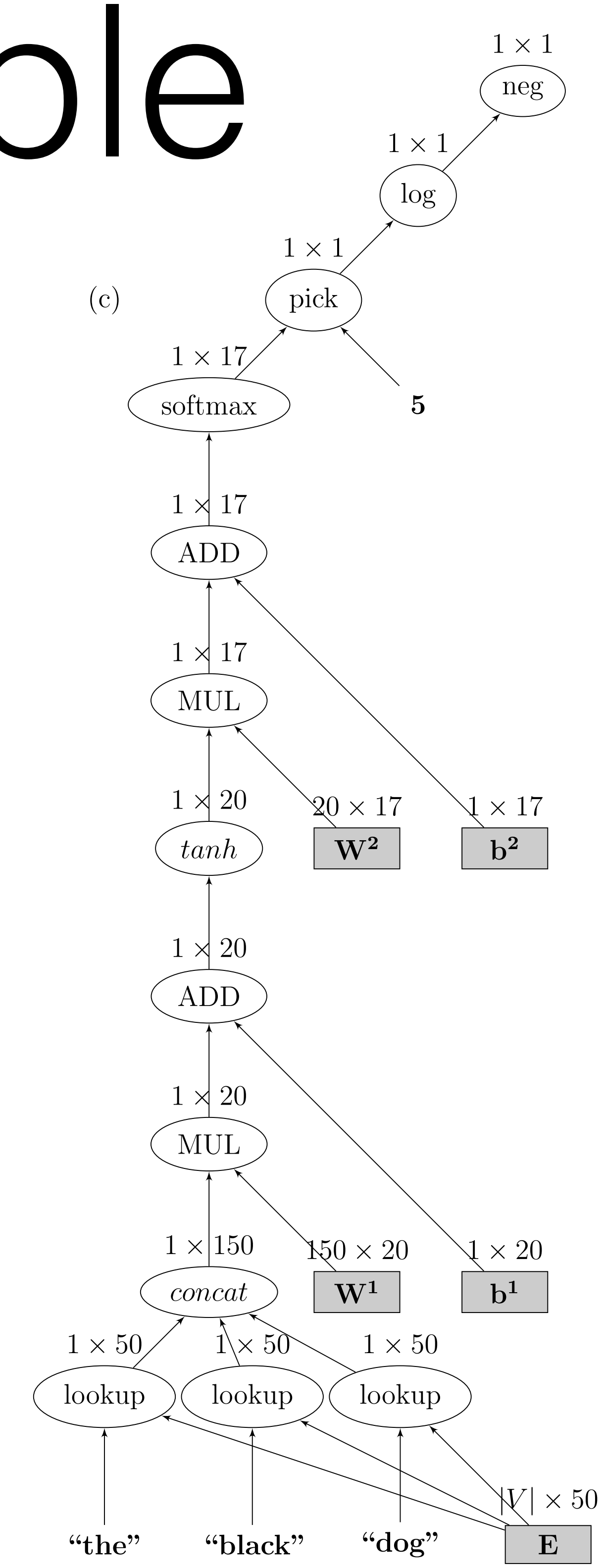
DyNet Example

```
# model initialization.
model = Model()
mW1 = model.add_parameters((20,150))
mb1 = model.add_parameters(20)
mW2 = model.add_parameters((17,20))
mb2 = model.add_parameters(17)
lookup = model.add_lookup_parameters((100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(mW1)
b1 = parameter(mb1)
W2 = parameter(mW2)
b2 = parameter(mb2)
def get_index(x): return 1
# Generate the embeddings layer.
vthe   = lookup[get_index("the")]
vblack = lookup[get_index("black")]
vdog   = lookup[get_index("dog")]

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))

loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
```



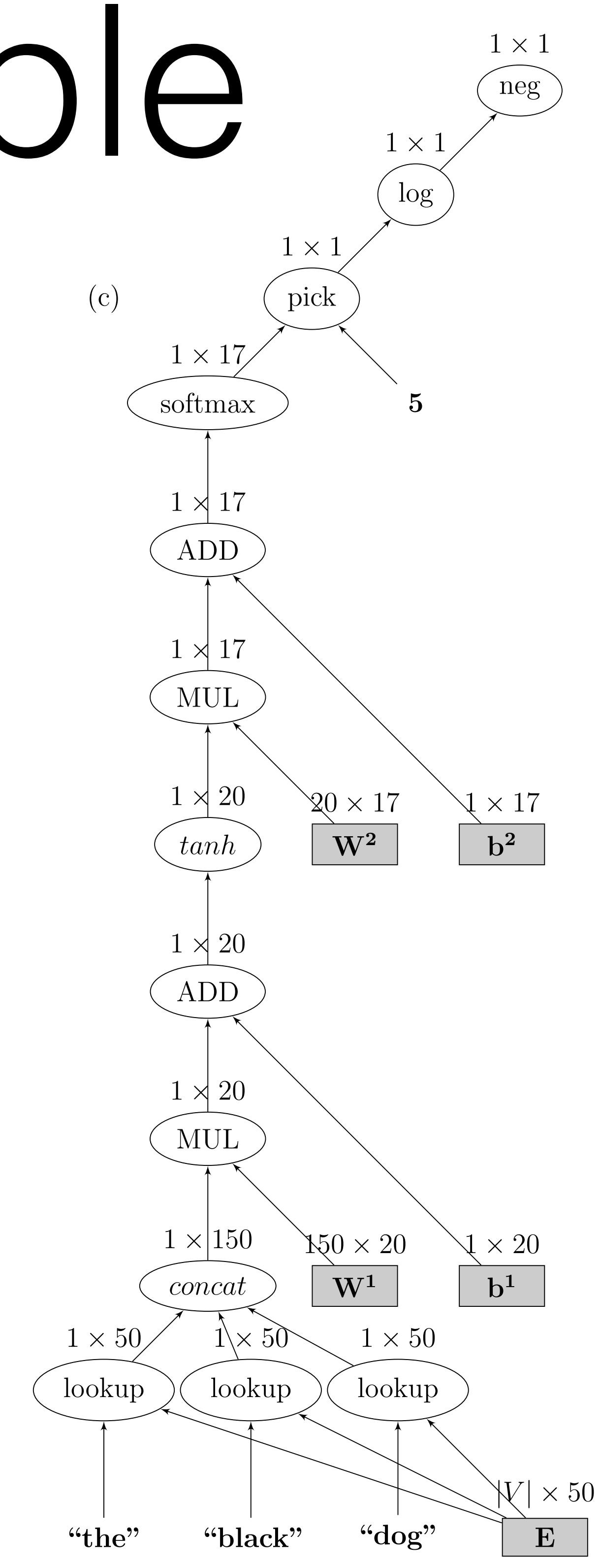
DyNet Example

```
# model initialization.
model = Model()
mW1 = model.add_parameters((20,150))
mb1 = model.add_parameters(20)
mW2 = model.add_parameters((17,20))
mb2 = model.add_parameters(17)
lookup = model.add_lookup_parameters((100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(mW1)
b1 = parameter(mb1)
W2 = parameter(mW2)
b2 = parameter(mb2)
def get_index(x): return 1
# Generate the embeddings layer.
vthe   = lookup[get_index("the")]
vblack = lookup[get_index("black")]
vdog   = lookup[get_index("dog")]

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))

loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
```



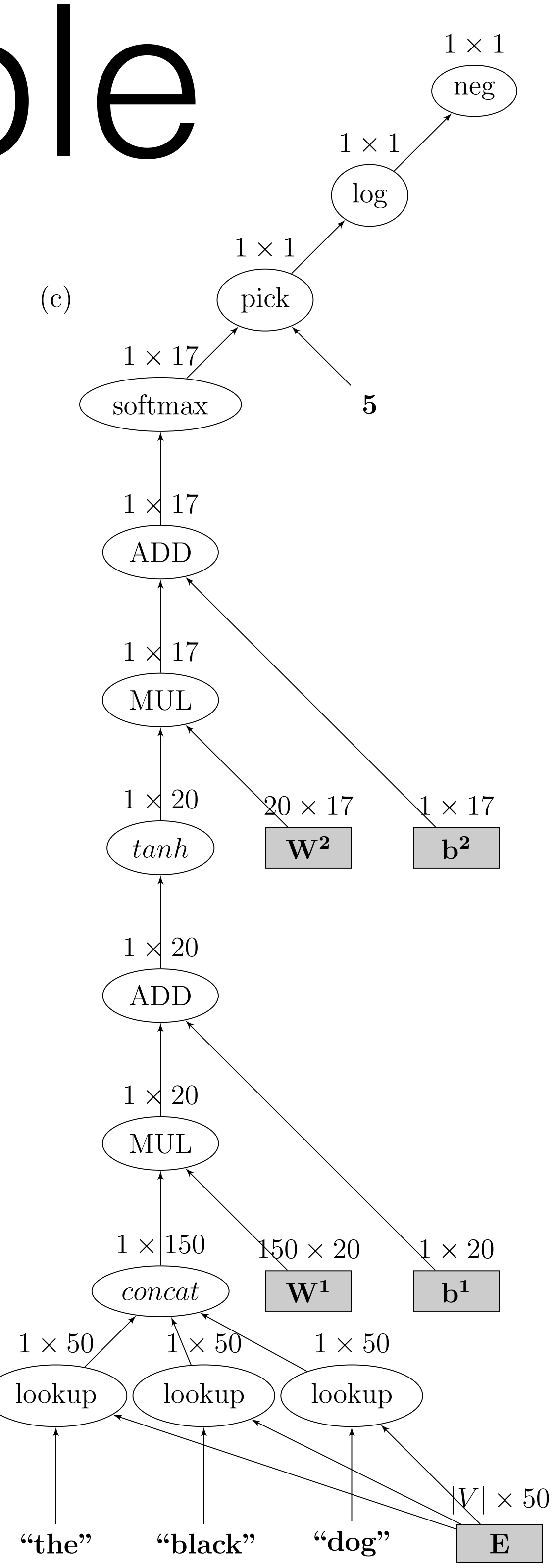
DyNet Example

```
# model initialization.
model = Model()
mW1 = model.add_parameters((20,150))
mb1 = model.add_parameters(20)
mW2 = model.add_parameters((17,20))
mb2 = model.add_parameters(17)
lookup = model.add_lookup_parameters((100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(mW1)
b1 = parameter(mb1)
W2 = parameter(mW2)
b2 = parameter(mb2)
def get_index(x): return 1
# Generate the embeddings layer.
vthe   = lookup[get_index("the")]
vblack = lookup[get_index("black")]
vdog   = lookup[get_index("dog")]

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))

loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
```



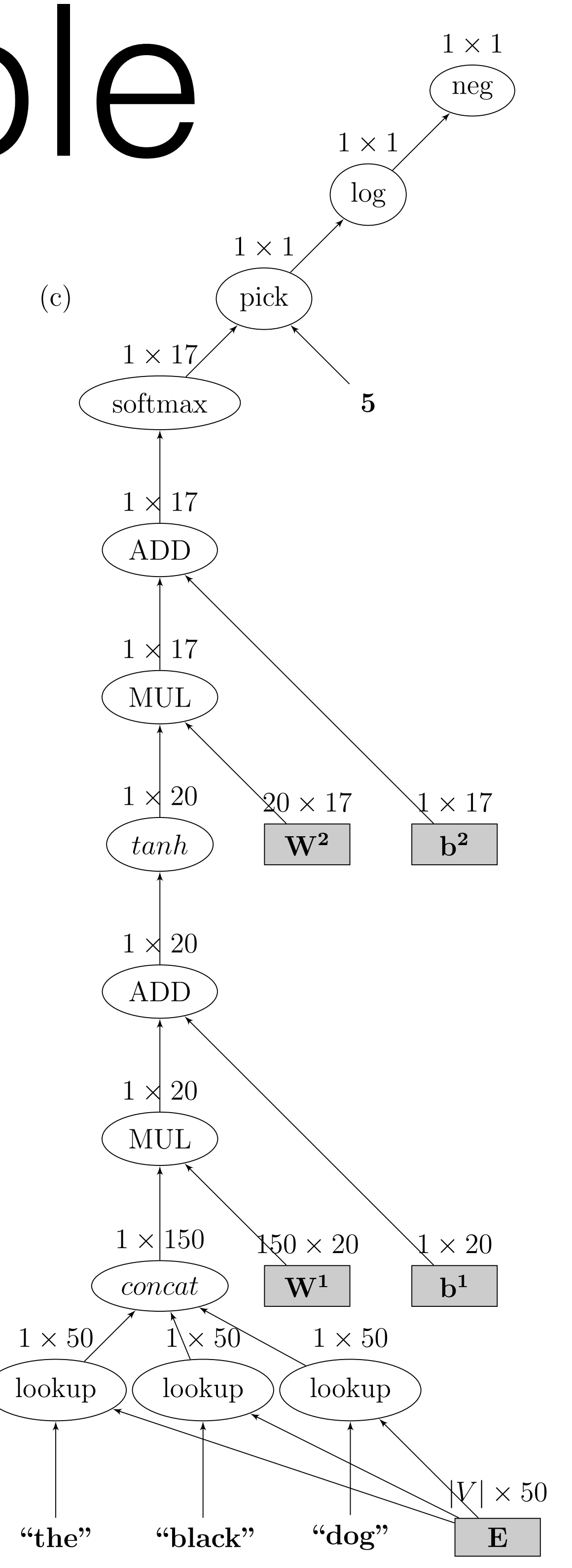
DyNet Example

```
# model initialization.
model = Model()
mW1 = model.add_parameters((20,150))
mb1 = model.add_parameters(20)
mW2 = model.add_parameters((17,20))
mb2 = model.add_parameters(17)
lookup = model.add_lookup_parameters((100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(mW1)
b1 = parameter(mb1)
W2 = parameter(mW2)
b2 = parameter(mb2)
def get_index(x): return 1
# Generate the embeddings layer.
vthe   = lookup[get_index("the")]
vblack = lookup[get_index("black")]
vdog   = lookup[get_index("dog")]

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))

loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
```



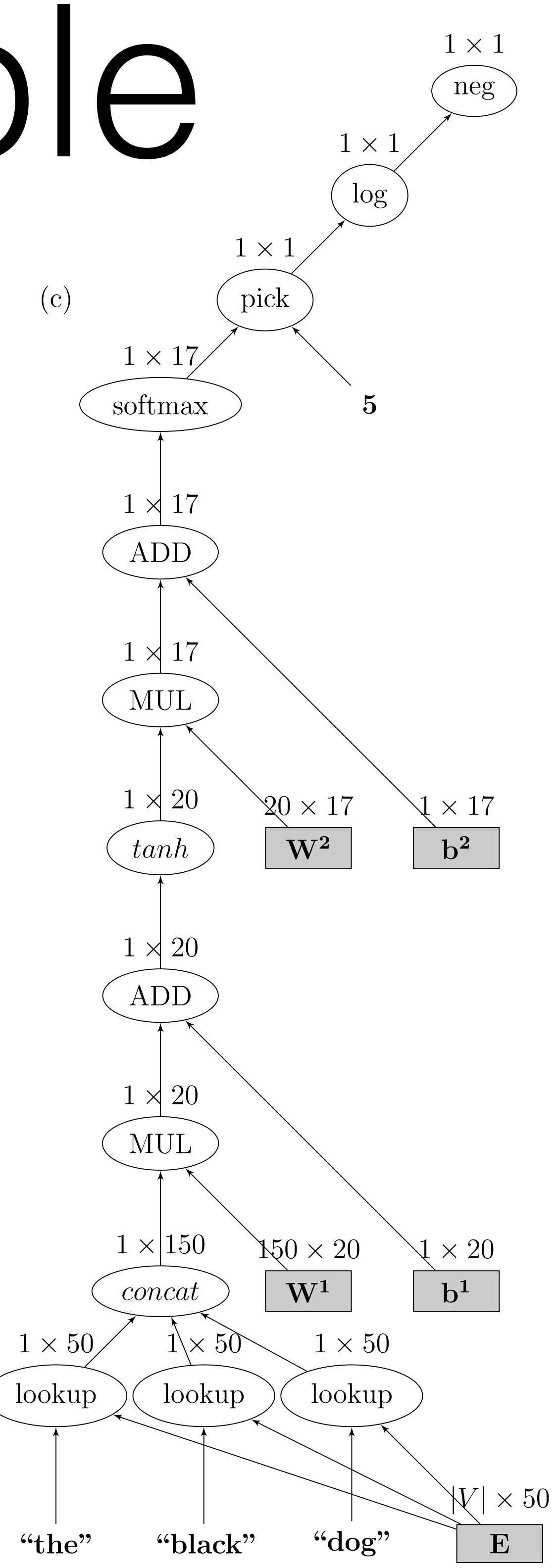
DyNet Example

```
# model initialization.
model = Model()
mW1 = model.add_parameters((20,150))
mb1 = model.add_parameters(20)
mW2 = model.add_parameters((17,20))
mb2 = model.add_parameters(17)
lookup = model.add_lookup_parameters((100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(mW1)
b1 = parameter(mb1)
W2 = parameter(mW2)
b2 = parameter(mb2)
def get_index(x): return 1
# Generate the embeddings layer.
vthe   = lookup[get_index("the")]
vblack = lookup[get_index("black")]
vdog   = lookup[get_index("dog")]

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))
```

```
loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
```



Questions?