10-301/10-601 Introduction to Machine Learning

Machine Learning Department
School of Computer Science
Carnegie Mellon University

Automatic Differentiation
&
Transformers

Matt Gormley
Lecture 28
Nov. 1, 2023

Reminders

- Homework 6: Learning Theory & Generative Models
  - Out: Fri, Oct 27
  - Due: Fri, Nov 3 at 11:59pm
- Exam 2 Practice Problems
  - Out: Fri, Nov 3
- Exam 2: Thu, Nov 9

MODULE-BASED AUTOMATIC DIFFERENTIATION

Backpropagation

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

<u>Forward Computation</u>

1. Write an **algorithm** for evaluating the function y = f(**x**). The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the "**computation graph**")
2. Visit each node in **topological order**.
   For variable u_i with inputs $v_1, \ldots, v_N$
   a. Compute $u_i = g_i(v_1, \ldots, v_N)$
   b. Store the result at the node

<u>Backward Computation (Version A)</u>

1. **Initialize** dy/dy = 1.
2. Visit each node v_j in **reverse topological order**.
   Let $u_1, \ldots, u_M$ denote all the nodes with v_j as an input
   Assuming that $y = h(\mathbf{u}) = h(u_1, \ldots, u_M)$
   and $\mathbf{u} = g(\mathbf{v})$ or equivalently $u_i = g_i(v_1, \ldots, v_j, \ldots, v_N)$ for all i
   a. We already know dy/du_i for all i
   b. Compute dy/dv_j as below (Choice of algorithm ensures computing (du_i/dv_j) is easy)

$$\frac{dy}{dv_j} = \sum_{i=1}^{M} \frac{dy}{du_i} \frac{du_i}{dv_j}$$

Return partial derivatives dy/du_i for all variables

Training

Backpropagation

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

<u>Forward Computation</u>

1. Write an **algorithm** for evaluating the function y = f(**x**). The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the "**computation graph**")
2. Visit each node in **topological order**.
   For variable u_i with inputs $v_1, \dots, v_N$
   a. Compute $u_i = g_i(v_1, \dots, v_N)$
   b. Store the result at the node

<u>Backward Computation (Version B)</u>

1. **Initialize** all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
2. Visit each node in **reverse topological order**.
   For variable $u_i = g_i(v_1, \dots, v_N)$
   a. We already know dy/du_i
   b. Increment dy/dv_j by $(dy/du_i)(du_i/dv_j)$
   (Choice of algorithm ensures computing (du_i/dv_j) is easy)

Return partial derivatives dy/du_i for all variables

Backpropagation

Why is the backpropagation algorithm efficient?

1. Reuses **computation from the forward pass** in the backward pass
2. Reuses **partial derivatives** throughout the backward pass (*but only if the algorithm reuses shared computation in the forward pass*)

   (Key idea: partial derivatives in the backward pass should be thought of as variables stored for reuse)

Background

A Recipe for

Gradients

Backpropagation can compute this gradient!

And it's a **special case of a more general algorithm** called reverse-mode automatic differentiation that can compute the gradient of any differentiable function efficiently!

1. Given training dat

$$\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N$$

$\boldsymbol{y}_i)$

2. Choose each of t

- Decision functio

$$\hat{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$$

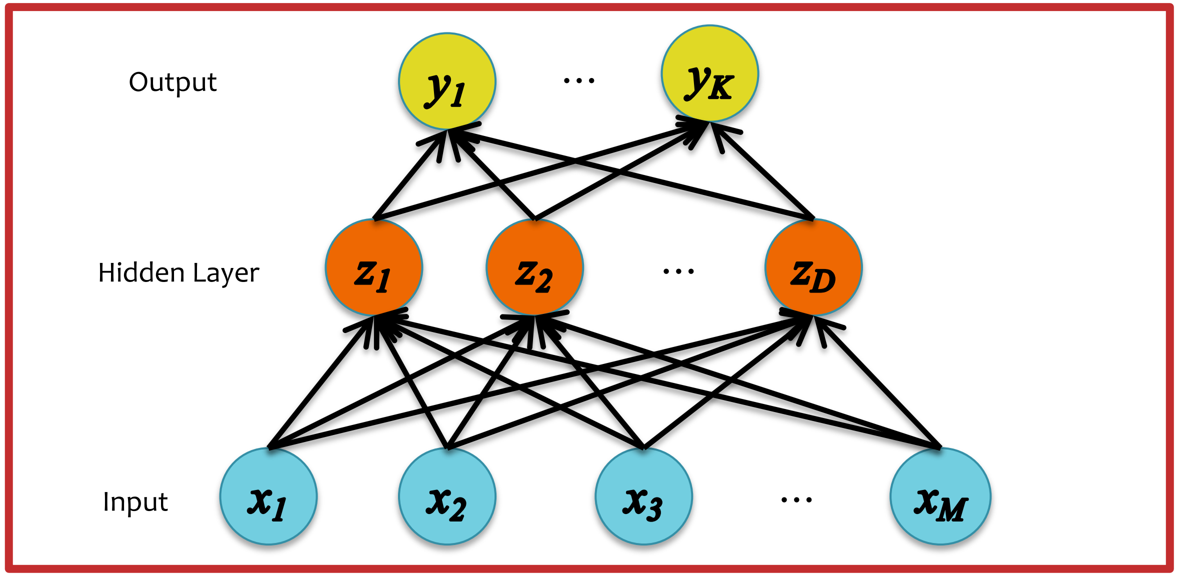
- Loss function

$$\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}_i) \in \mathbb{R}$$

opposite the gradient)

$$\boldsymbol{\theta}^{(t} \quad {}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

Backpropagation: Abstract Picture



Forward

5. $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$

4. $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$

3. $\mathbf{b} = \boldsymbol{\beta}\mathbf{z}$

2. $\mathbf{z} = \sigma(\mathbf{a})$

1. $\mathbf{a} = \boldsymbol{\alpha}\mathbf{x}$

Backward

6. $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$

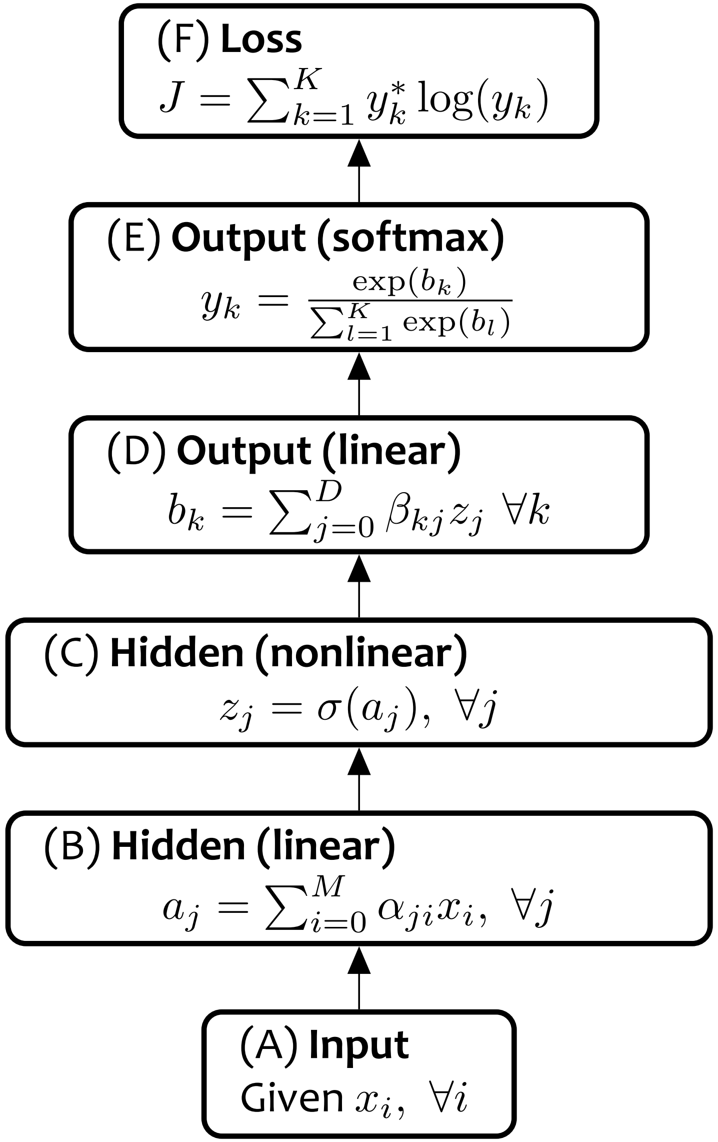
7. $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T \left(\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T\right)$

8. $\mathbf{g}_{\boldsymbol{\beta}} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$

$\mathbf{g}_{\mathbf{z}} = \boldsymbol{\beta}^T \mathbf{g}_{\mathbf{b}}^T$

10. $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$

11. $\mathbf{g}_{\boldsymbol{\alpha}} = \mathbf{g}_{\mathbf{a}} \mathbf{x}^T$



(F) **Loss**
$J = \sum_{k=1}^K y_k^* \log(y_k)$

(E) **Output (softmax)**
$y_k = \frac{\exp(b_k)}{\sum_{l=1}^K \exp(b_l)}$

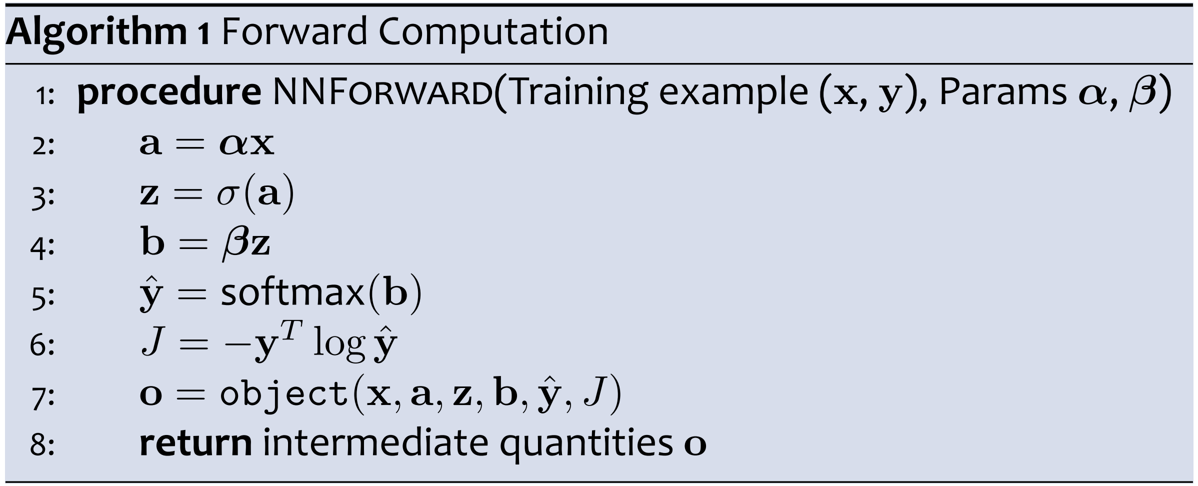
(D) **Output (linear)**
$b_k = \sum_{j=0}^D \beta_{kj} z_j \;\; \forall k$

(C) **Hidden (nonlinear)**
$z_j = \sigma(a_j), \;\; \forall j$

(B) **Hidden (linear)**
$a_j = \sum_{i=0}^M \alpha_{ji} x_i, \;\; \forall j$

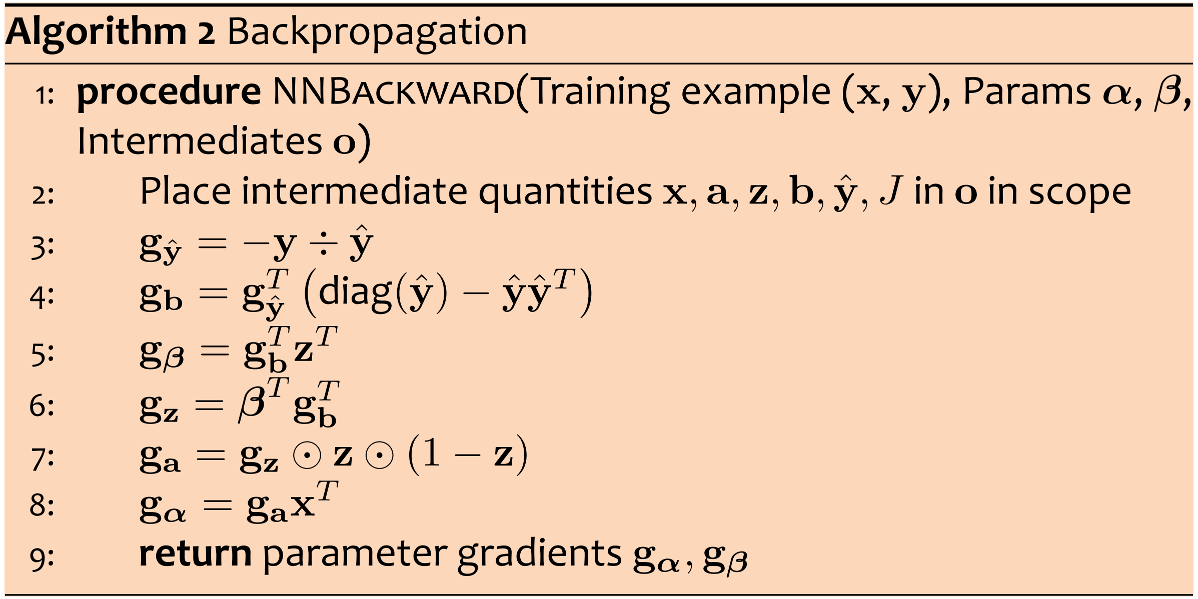
(A) **Input**
Given $x_i, \;\; \forall i$

Backpropagation: Procedural Method



Algorithm 1 Forward Computation

```
procedure NNFORWARD(Training example $(\mathbf{x}, \mathbf{y})$, Params $\boldsymbol{\alpha}, \boldsymbol{\beta}$)
    $\mathbf{a} = \boldsymbol{\alpha}\mathbf{x}$
    $\mathbf{z} = \sigma(\mathbf{a})$
    $\mathbf{b} = \boldsymbol{\beta}\mathbf{z}$
    $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$
    $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$
    $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$
    return intermediate quantities $\mathbf{o}$
```



Algorithm 2 Backpropagation

```
procedure NNBACKWARD(Training example $(\mathbf{x}, \mathbf{y})$, Params $\boldsymbol{\alpha}, \boldsymbol{\beta}$, Intermediates $\mathbf{o}$)
    Place intermediate quantities $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$ in $\mathbf{o}$ in scope
    $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$
    $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T \left(\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T\right)$
    $\mathbf{g}_{\boldsymbol{\beta}} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$
    $\mathbf{g}_{\mathbf{z}} = \boldsymbol{\beta}^T \mathbf{g}_{\mathbf{b}}^T$
    $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$
    $\mathbf{g}_{\boldsymbol{\alpha}} = \mathbf{g}_{\mathbf{a}} \mathbf{x}^T$
    return parameter gradients $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}$
```

Drawbacks of Procedural Method

1. Hard to reuse / adapt for other models
2. (Possibly) harder to make individual steps more efficient
3. Hard to find source of error if finite-difference check reports an error (since it tells you only that there is an error somewhere in those 17 lines of code)

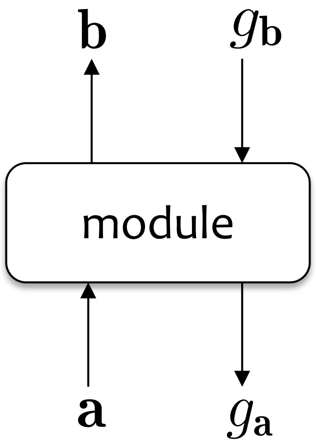
Module-based AutoDiff

Module-based automatic differentiation (AD / Autodiff) is a technique that has long been used to develop libraries for deep learning

- **Dynamic neural network packages** allow a specification of the computation graph dynamically at runtime
  - PyTorch http://pytorch.org
  - Torch http://torch.ch
  - DyNet https://dynet.readthedocs.io
  - TensorFlow with Eager Execution https://www.tensorflow.org
- **Static neural network packages** require a static specification of a computation graph which is subsequently compiled into code
  - TensorFlow with Graph Execution https://www.tensorflow.org
  - Aesara (and Theano) https://aesara.readthedocs.io
  - *(These libraries are also module-based, but herein by "module-based AD" we mean the dynamic approach)*

Module-based AutoDiff

- **Key Idea:**
  - componentize the computation of the neural-network into layers
  - each layer consolidates multiple **real-valued nodes** in the computation graph (a subset of them) into one **vector-valued node** (aka. a **module**)
- Each **module** is capable of two actions:
  1. Forward computation of output $\mathbf{b} = [b_1, \ldots, b_B]$ given input $\mathbf{a} = [a_1, \ldots, a_A]$ via some differentiable function f. That is $\mathbf{b} = f(\mathbf{a})$.
  2. Backward computation of the gradient of the input $g_\mathbf{a} = \nabla_\mathbf{a} J = [\frac{\partial J}{\partial a_1}, \ldots, \frac{\partial J}{\partial a_A}]$ given the gradient of output $g_\mathbf{b} = \nabla_\mathbf{b} J = [\frac{\partial J}{\partial b_1}, \ldots, \frac{\partial J}{\partial b_B}]$, where J is the final real-valued output of the entire computation graph. This is done via the chain rule $\frac{\partial J}{\partial a_i} = \sum_{j=1}^{J} \frac{\partial J}{\partial b_j} \frac{d b_j}{d a_i}$ for all $i \in \{1, \ldots, A\}$.



Module-based AutoDiff

Dimensions: input $\mathbf{a} \in \mathbb{R}^A$, output $\mathbf{b} \in \mathbb{R}^B$, gradient of output $\mathbf{g_a} \triangleq \nabla_\mathbf{a} J \in \mathbb{R}^A$, and gradient of input $\mathbf{g_b} \triangleq \nabla_\mathbf{b} J \in \mathbb{R}^B$.

Sigmoid Module The sigmoid layer has only one input vector $\mathbf{a}$. Below σ is the sigmoid applied element-wise, and $\odot$ is element-wise multiplication s.t. $\mathbf{u} \odot \mathbf{v} = [u_1 v_1, \ldots, u_M v_M]$.

```
procedure SigmoidForward($\mathbf{a}$)
    $\mathbf{b} = \sigma(\mathbf{a})$
    return $\mathbf{b}$
procedure SigmoidBackward($\mathbf{a}$, $\mathbf{b}$, $\mathbf{g_b}$)
    $\mathbf{g_a} = \mathbf{g_b} \odot \mathbf{b} \odot (1 - \mathbf{b})$
    return $\mathbf{g_a}$
```

Softmax Module The softmax layer has only one input vector $\mathbf{a}$. For any vector $\mathbf{v} \in \mathbb{R}^D$, we have that diag$(\mathbf{v})$ returns a $D \times D$ diagonal matrix whose diagonal entries are $v_1, v_2, \ldots, v_D$ and whose non-diagonal entries are zero.

```
procedure SoftmaxForward($\mathbf{a}$)
    $\mathbf{b} = \text{softmax}(\mathbf{a})$
    return $\mathbf{b}$
procedure SoftmaxBackward($\mathbf{a}$, $\mathbf{b}$, $\mathbf{g_b}$)
    $\mathbf{g_a} = \mathbf{g}_\mathbf{b}^T \left(\text{diag}(\mathbf{b}) - \mathbf{b}\mathbf{b}^T\right)$
    return $\mathbf{g_a}$
```

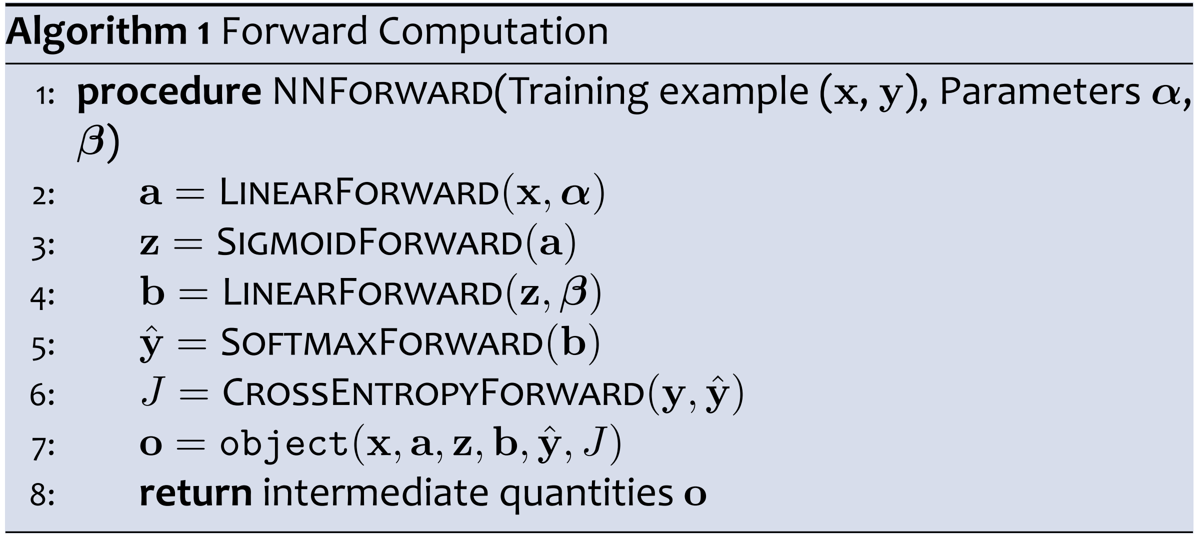
Linear Module The linear layer has two inputs: a vector $\mathbf{a}$ and parameters $\omega \in \mathbb{R}^{B \times A}$. The output $\mathbf{b}$ is not used by LinearBackward, but we pass it in for consistency of form.

```
procedure LinearForward($\mathbf{a}$, $\boldsymbol{\omega}$)
    $\mathbf{b} = \boldsymbol{\omega}\mathbf{a}$
    return $\mathbf{b}$
procedure LinearBackward($\mathbf{a}$, $\omega$, $\mathbf{b}$, $\mathbf{g_b}$)
    $\mathbf{g}_{\boldsymbol{\omega}} = \mathbf{g_b}\mathbf{a}^T$
    $\mathbf{g_a} = \boldsymbol{\omega}^T \mathbf{g_b}$
    return $\mathbf{g}_{\boldsymbol{\omega}}, \mathbf{g_a}$
```

Cross-Entropy Module The cross-entropy layer has two inputs: a gold one-hot vector $\mathbf{a}$ and a predicted probability distribution $\hat{\mathbf{a}}$. It's output $b \in \mathbb{R}$ is a scalar. Below $\div$ is element-wise division. The output b is not used by CrossEntropyBackward, but we pass it in for consistency of form.

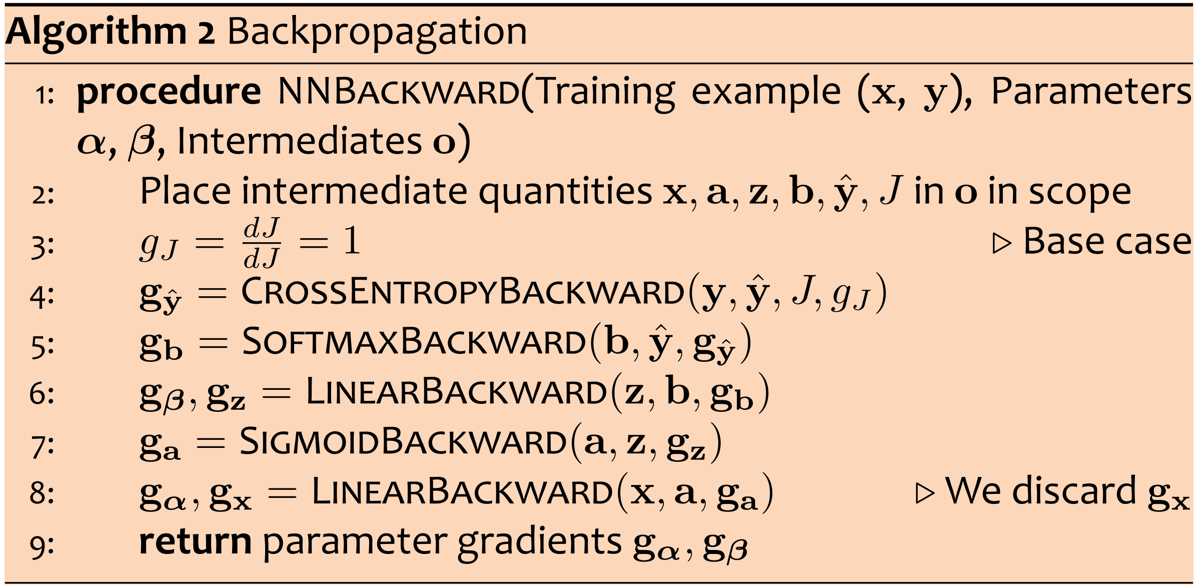
```
procedure CrossEntropyForward($\mathbf{a}$, $\hat{\mathbf{a}}$)
    $b = -\mathbf{a}^T \log \hat{\mathbf{a}}$
    return $\mathbf{b}$
procedure CrossEntropyBackward($\mathbf{a}$, $\hat{\mathbf{a}}$, $b$, $g_b$)
    $\mathbf{g}_{\hat{\mathbf{a}}} = -g_b(\mathbf{a} \div \hat{\mathbf{a}})$
    return $\mathbf{g_a}$
```

Module-based AutoDiff



Algorithm 1 Forward Computation

```
procedure NNFORWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)
    $\mathbf{a} = \text{LINEARFORWARD}(\mathbf{x}, \boldsymbol{\alpha})$
    $\mathbf{z} = \text{SIGMOIDFORWARD}(\mathbf{a})$
    $\mathbf{b} = \text{LINEARFORWARD}(\mathbf{z}, \boldsymbol{\beta})$
    $\hat{\mathbf{y}} = \text{SOFTMAXFORWARD}(\mathbf{b})$
    $J = \text{CROSSENTROPYFORWARD}(\mathbf{y}, \hat{\mathbf{y}})$
    $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$
    return intermediate quantities $\mathbf{o}$
```



Algorithm 2 Backpropagation

```
procedure NNBACKWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, Intermediates $\mathbf{o}$)
    Place intermediate quantities $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$ in $\mathbf{o}$ in scope
    $g_J = \frac{dJ}{dJ} = 1$                                        ▷ Base case
    $\mathbf{g}_{\hat{\mathbf{y}}} = \text{CROSSENTROPYBACKWARD}(\mathbf{y}, \hat{\mathbf{y}}, J, g_J)$
    $\mathbf{g}_{\mathbf{b}} = \text{SOFTMAXBACKWARD}(\mathbf{b}, \hat{\mathbf{y}}, \mathbf{g}_{\hat{\mathbf{y}}})$
    $\mathbf{g}_{\boldsymbol{\beta}}, \mathbf{g}_{\mathbf{z}} = \text{LINEARBACKWARD}(\mathbf{z}, \mathbf{b}, \mathbf{g}_{\mathbf{b}})$
    $\mathbf{g}_{\mathbf{a}} = \text{SIGMOIDBACKWARD}(\mathbf{a}, \mathbf{z}, \mathbf{g}_{\mathbf{z}})$
    $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\mathbf{x}} = \text{LINEARBACKWARD}(\mathbf{x}, \mathbf{a}, \mathbf{g}_{\mathbf{a}})$        ▷ We discard $\mathbf{g}_{\mathbf{x}}$
    return parameter gradients $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}$
```

Advantages of Module-based AutoDiff

1. Easy to reuse / adapt for other models
2. Encapsulated layers are easier to optimize (e.g. implement in C++ or CUDA)
3. Easier to find bugs because we can run a finite-difference check on each layer separately

Module-based AutoDiff (OOP Version)

Object-Oriented Implementation:

- Let each module be an **object**
- Then allow the **control flow** dictate the creation of the **computation graph**
- No longer need to implement NNBackward(·), just follow the computation graph in **reverse topological order**

```
class Sigmoid(Module)
    method forward($\mathbf{a}$)
        $\mathbf{b} = \sigma(\mathbf{a})$
        return $\mathbf{b}$
    method backward($\mathbf{a}$, $\mathbf{b}$, $\mathbf{g_b}$)
        $\mathbf{g_a} = \mathbf{g_b} \odot \mathbf{b} \odot (1 - \mathbf{b})$
        return $\mathbf{g_a}$
```

```
class Linear(Module)
    method forward($\mathbf{a}$, $\boldsymbol{\omega}$)
        $\mathbf{b} = \boldsymbol{\omega}\mathbf{a}$
        return $\mathbf{b}$
    method backward($\mathbf{a}$, $\boldsymbol{\omega}$, $\mathbf{b}$, $\mathbf{g_b}$)
        $\mathbf{g}_{\boldsymbol{\omega}} = \mathbf{g_b}\mathbf{a}^T$
        $\mathbf{g_a} = \boldsymbol{\omega}^T\mathbf{g_b}$
        return $\mathbf{g}_{\boldsymbol{\omega}}, \mathbf{g_a}$
```

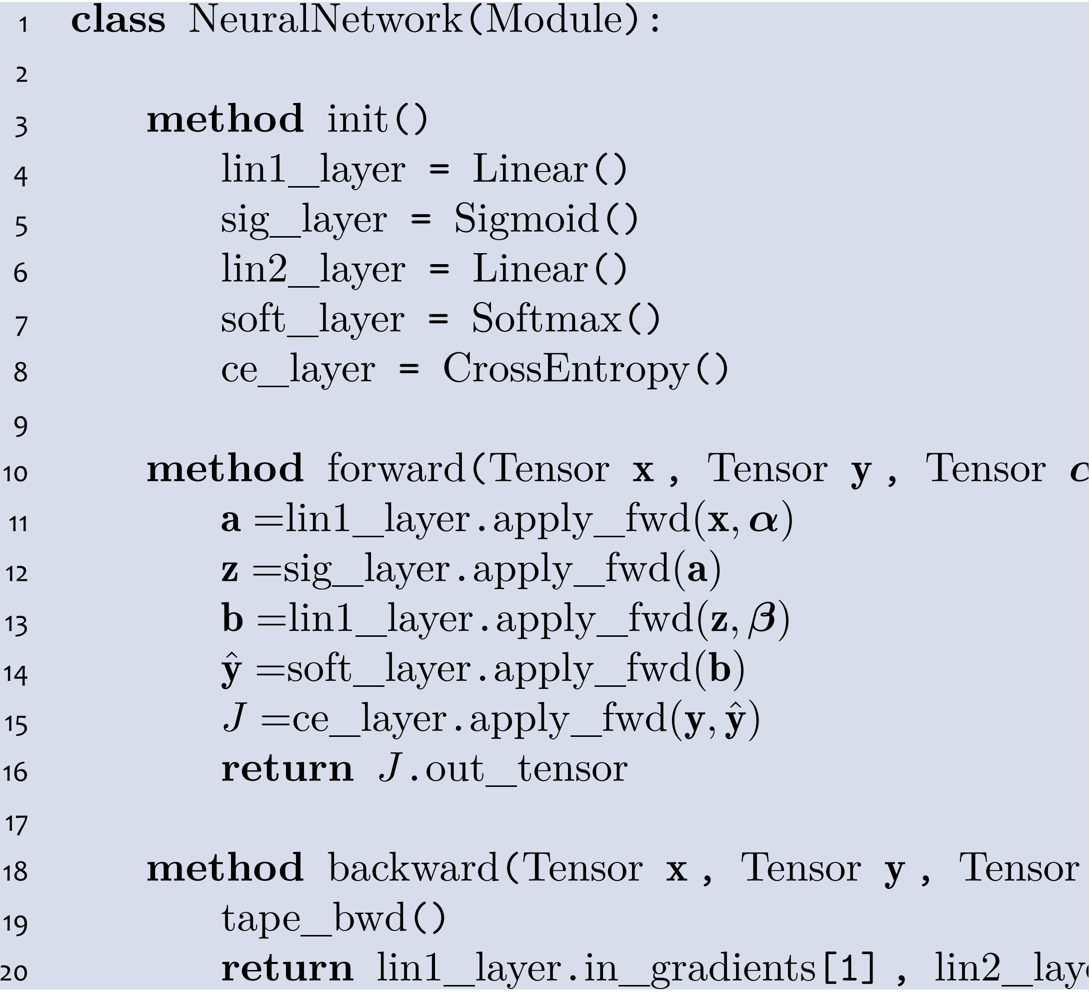
```
class Softmax(Module)
    method forward($\mathbf{a}$)
        $\mathbf{b} = \texttt{softmax}(\mathbf{a})$
        return $\mathbf{b}$
    method backward($\mathbf{a}$, $\mathbf{b}$, $\mathbf{g_b}$)
        $\mathbf{g_a} = \mathbf{g_b}^T \left(\texttt{diag}(\mathbf{b}) - \mathbf{b}\mathbf{b}^T\right)$
        return $\mathbf{g_a}$
```

```
class CrossEntropy(Module)
    method forward($\mathbf{a}$, $\hat{\mathbf{a}}$)
        $b = -\mathbf{a}^T \log \hat{\mathbf{a}}$
        return $\mathbf{b}$
    method backward($\mathbf{a}$, $\hat{\mathbf{a}}$, $b$, $g_b$)
        $\mathbf{g}_{\hat{\mathbf{a}}} = -g_b(\mathbf{a} \div \hat{\mathbf{a}})$
        return $\mathbf{g_a}$
```

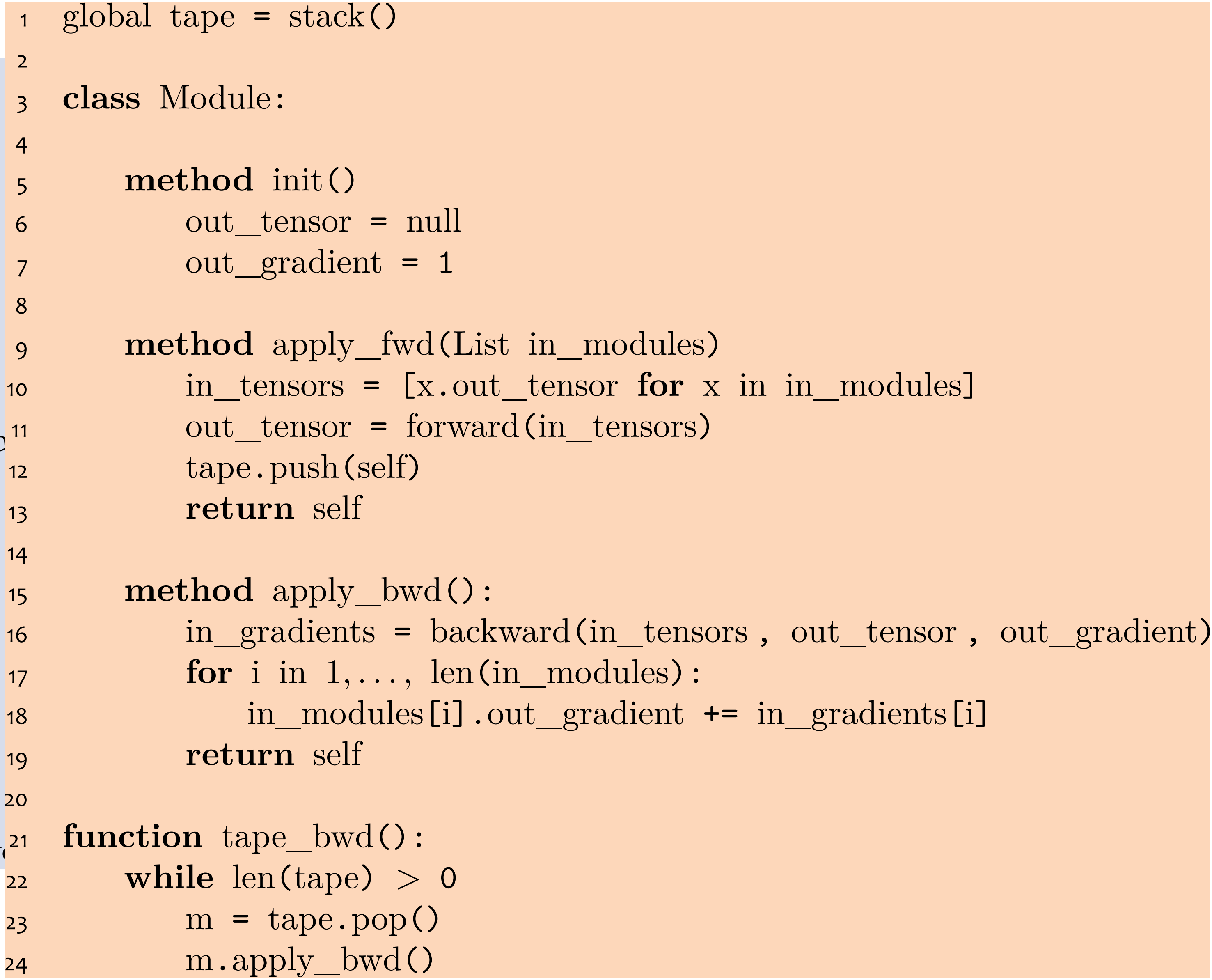
Module-based AutoDiff (OOP Version)

```
class NeuralNetwork(Module):

    method init()
        lin1_layer = Linear()
        sig_layer = Sigmoid()
        lin2_layer = Linear()
        soft_layer = Softmax()
        ce_layer = CrossEntropy()

    method forward(Tensor $\mathbf{x}$ , Tensor $\mathbf{y}$ , Tensor $\boldsymbol{\alpha}$ , Tensor $\boldsymbol{\beta}$)
        $\mathbf{a}$ =lin1_layer.apply_fwd($\mathbf{x}, \boldsymbol{\alpha}$)
        $\mathbf{z}$ =sig_layer.apply_fwd($\mathbf{a}$)
        $\mathbf{b}$ =lin1_layer.apply_fwd($\mathbf{z}, \boldsymbol{\beta}$)
        $\hat{\mathbf{y}}$ =soft_layer.apply_fwd($\mathbf{b}$)
        $J$ =ce_layer.apply_fwd($\mathbf{y}, \hat{\mathbf{y}}$)
        return $J$.out_tensor

    method backward(Tensor $\mathbf{x}$ , Tensor $\mathbf{y}$ , Tensor $\boldsymbol{\alpha}$ , Tensor $\boldsymbol{\beta}$)
        tape_bwd()
        return lin1_layer.in_gradients[1] , lin2_layer.in_gradients[1]
```

Module-based AutoDiff (OOP Version)



```
class NeuralNetwork(Module):

    method init()
        lin1_layer = Linear()
        sig_layer = Sigmoid()
        lin2_layer = Linear()
        soft_layer = Softmax()
        ce_layer = CrossEntropy()

    method forward(Tensor x , Tensor y , Tensor
        $\mathbf{a}$ =lin1_layer.apply_fwd($\mathbf{x}, \boldsymbol{\alpha}$)
        $\mathbf{z}$ =sig_layer.apply_fwd($\mathbf{a}$)
        $\mathbf{b}$ =lin1_layer.apply_fwd($\mathbf{z}, \boldsymbol{\beta}$)
        $\hat{\mathbf{y}}$ =soft_layer.apply_fwd($\mathbf{b}$)
        $J$ =ce_layer.apply_fwd($\mathbf{y}, \hat{\mathbf{y}}$)
        return $J$.out_tensor

    method backward(Tensor x , Tensor y , Tensor
        tape_bwd()
        return lin1_layer.in_gradients[1] , lin2_lay
```



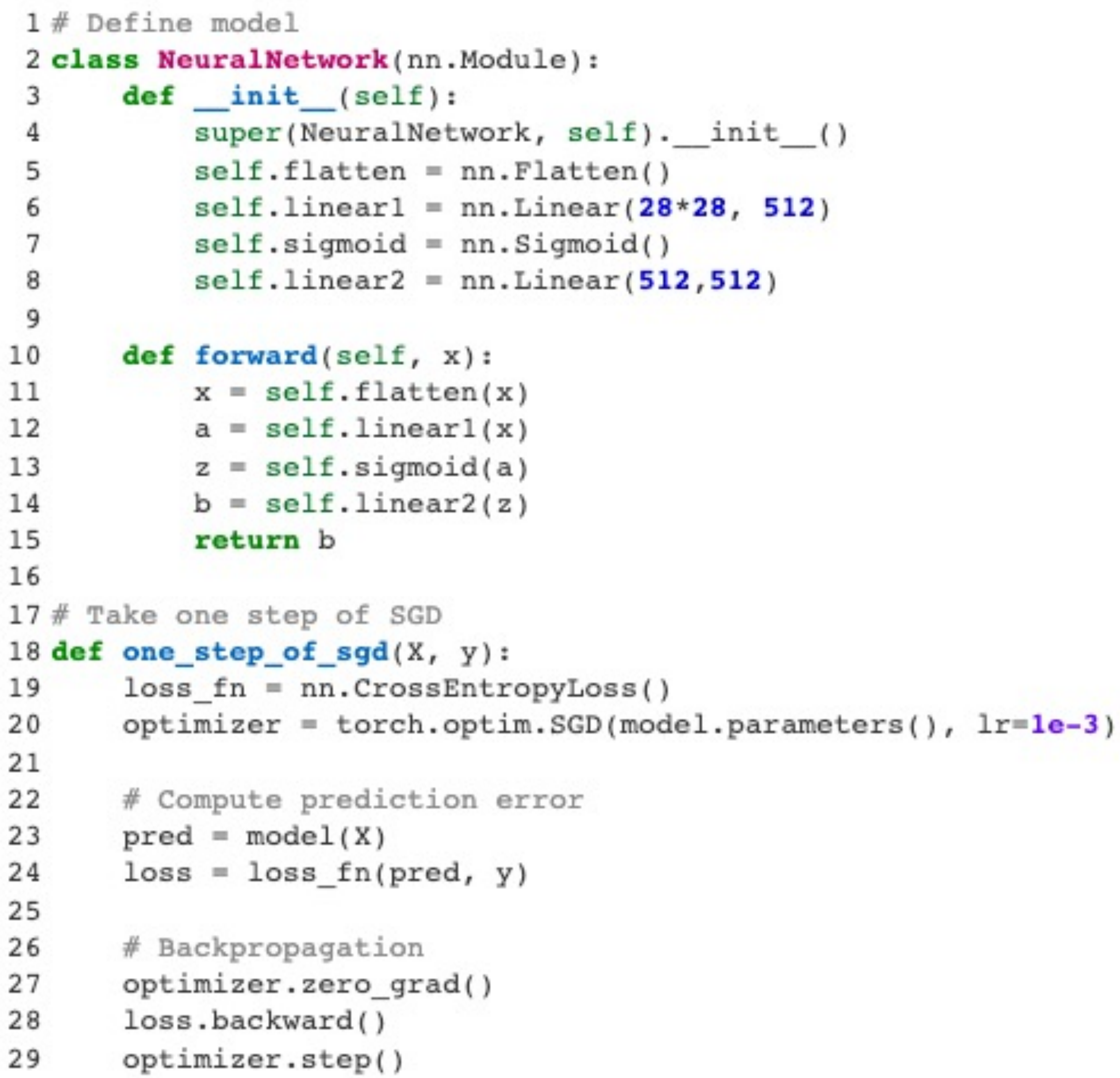
```
global tape = stack()

class Module:

    method init()
        out_tensor = null
        out_gradient = 1

    method apply_fwd(List in_modules)
        in_tensors = [x.out_tensor for x in in_modules]
        out_tensor = forward(in_tensors)
        tape.push(self)
        return self

    method apply_bwd():
        in_gradients = backward(in_tensors , out_tensor , out_gradient)
        for i in 1,..., len(in_modules):
            in_modules[i].out_gradient += in_gradients[i]
        return self

function tape_bwd():
    while len(tape) > 0
        m = tape.pop()
        m.apply_bwd()
```

Module-based AutoDiff (OOP Version)

```
global tape = stack()

class Module:

    method init()
        out_tensor = null
        out_gradient = 1

    method apply_fwd(List in_modules)
        in_tensors = [x.out_tensor for x in in_modules]
        out_tensor = forward(in_tensors)
        tape.push(self)
        return self

    method apply_bwd():
        in_gradients = backward(in_tensors , out_tensor , out_gradient)
        for i in 1,..., len(in_modules):
            in_modules[i].out_gradient += in_gradients[i]
        return self

function tape_bwd():
    while len(tape) > 0
        m = tape.pop()
        m.apply_bwd()
```

PyTorch

The same simple neural network we defined in pseudocode can also be defined in PyTorch.



```
# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(28*28, 512)
        self.sigmoid = nn.Sigmoid()
        self.linear2 = nn.Linear(512,512)

    def forward(self, x):
        x = self.flatten(x)
        a = self.linear1(x)
        z = self.sigmoid(a)
        b = self.linear2(z)
        return b

# Take one step of SGD
def one_step_of_sgd(X, y):
    loss_fn = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

    # Compute prediction error
    pred = model(X)
    loss = loss_fn(pred, y)

    # Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Example adapted from https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html

PyTorch

Q: Why don’t we call linear.forward() in PyTorch?

A: This is just syntactic sugar. There’s a special method in Python `__call__` that allows you to define what happens when you treat an object as if it were a function.

In other words, running the following:

```
    linear(x)
```

is equivalent to running:

```
    linear.__call__(x)
```

which in PyTorch is (nearly) the same as running:

```
    linear.forward(x)
```

This is because PyTorch defines every Module’s `__call__` method to be something like this:

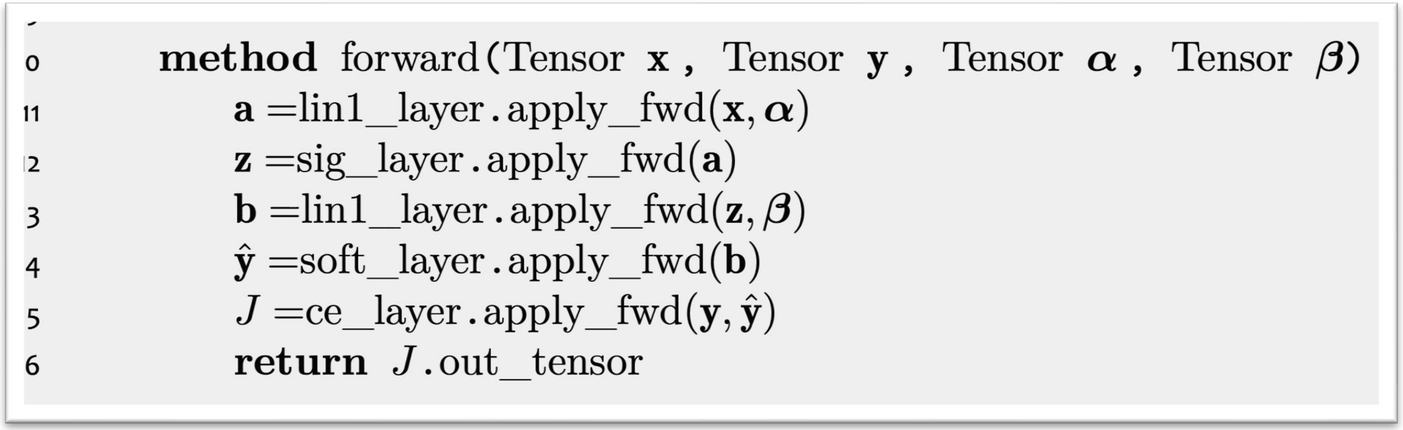
```
    def __call__(self):
        self.forward()
```

PyTorch

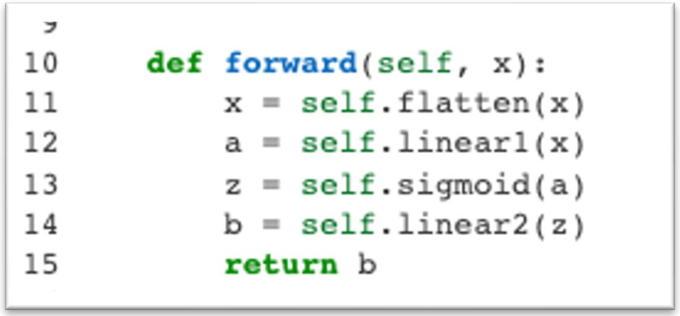
Q: Why don’t we pass in the parameters to a PyTorch Module?

A: This just makes your code cleaner.

In PyTorch, you store the parameters inside the Module and “mark” them as parameters that should contribute to the eventual gradient used by an optimizer



```
method forward(Tensor $\mathbf{x}$, Tensor $\mathbf{y}$, Tensor $\boldsymbol{\alpha}$, Tensor $\boldsymbol{\beta}$)
    $\mathbf{a}$ = lin1_layer.apply_fwd($\mathbf{x}, \boldsymbol{\alpha}$)
    $\mathbf{z}$ = sig_layer.apply_fwd($\mathbf{a}$)
    $\mathbf{b}$ = lin1_layer.apply_fwd($\mathbf{z}, \boldsymbol{\beta}$)
    $\hat{\mathbf{y}}$ = soft_layer.apply_fwd($\mathbf{b}$)
    $J$ = ce_layer.apply_fwd($\mathbf{y}, \hat{\mathbf{y}}$)
    return $J$.out_tensor
```



```
def forward(self, x):
    x = self.flatten(x)
    a = self.linear1(x)
    z = self.sigmoid(a)
    b = self.linear2(z)
    return b
```

LARGE LANGUAGE MODELS

What is ChatGPT?

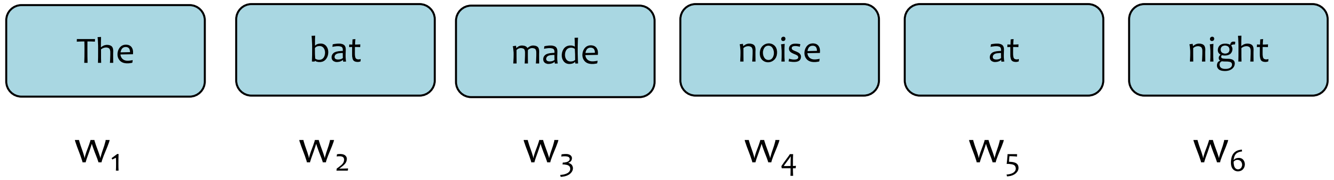
- ChatGPT is a large (in the sense of having many parameters) language model, fine-tuned to be a dialogue agent
- The base language model is GPT-3.5 which was trained on a large quantity of text

TASK: LANGUAGE MODELING

Recall…

n-Gram Language Model

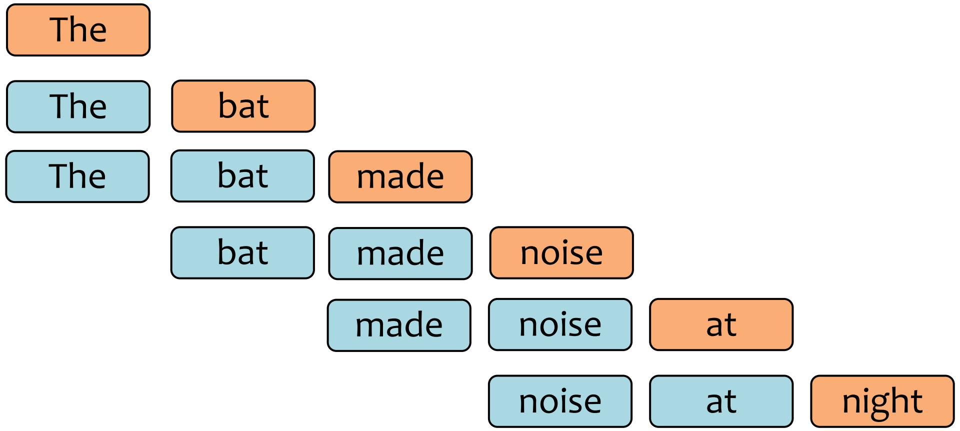
Question: How can we **define** a probability distribution over a sequence of length T?



n-Gram Model (n=3)

$$p(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} p(w_t \mid w_{t-1}, w_{t-2})$$

$p(w_1, w_2, w_3, \ldots , w_6) =$



$p(w_1)$

$p(w_2 \mid w_1)$

$p(w_3 \mid w_2, w_1)$

$p(w_4 \mid w_3, w_2)$

$p(w_5 \mid w_4, w_3)$

$p(w_6 \mid w_5, w_4)$

n-Gram Language Model

Recall…

Question: How can we **define** a probability distribution over a sequence of length T?

The	bat	made	noise	at	night
w_1	w_2	w_3	w_4	w_5	w_6

n-Gram Model (n=3)

$$p(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} p(w_t \mid w_{t-1}, w_{t-2})$$

$p(w_1, w_2, w_3, \ldots, w_6) =$

The

$p(w_1)$

The bat

$p(w_2 \mid w_1)$

The

Note: This is called a **model** because we made some **assumptions** about how many previous words to condition on (i.e. only n-1 words)

Learning an n-Gram Model

Recall…

Question: How do we **learn** the probabilities for the n-Gram Model?

$p(w_t \mid w_{t-2} = \text{The}, w_{t-1} = \text{bat})$

w_t	$p(\cdot \mid \cdot, \cdot)$
ate	0.015
…	
flies	0.046
…	
zebra	0.000

$p(w_t \mid w_{t-2} = \text{made}, w_{t-1} = \text{noise})$

w_t	$p(\cdot \mid \cdot, \cdot)$
at	0.020
…	
pollution	0.030
…	
zebra	0.000

$p(w_t \mid w_{t-2} = \text{cows}, w_{t-1} = \text{eat})$

w_t	$p(\cdot \mid \cdot, \cdot)$
corn	0.420
…	
grass	0.510
…	
zebra	0.000

Recall…

Learning an n-Gram Model

Question: How do we **learn** the probabilities for the n-Gram Model?

Answer: From data! Just **count** n-gram frequencies

…the **cows eat** **grass**…

…our **cows eat hay** daily…

…factory-farm **cows eat** **corn**…

…on an organic farm, **cows eat hay** and…

…do your **cows eat** **grass** or corn?...

…what do **cows eat if** they have…

…**cows eat** **corn** when there is no…

…which **cows eat which** foods depends…

…if **cows eat** **grass**…

…when **cows eat** **corn** their stomachs…

…should we let **cows eat** **corn**?...

$p(w_t \mid w_{t-2} = \text{cows}, w_{t-1} = \text{eat})$

w_t	$p(\cdot \mid \cdot, \cdot)$
corn	4/11
grass	3/11
hay	2/11
if	1/11
which	1/11

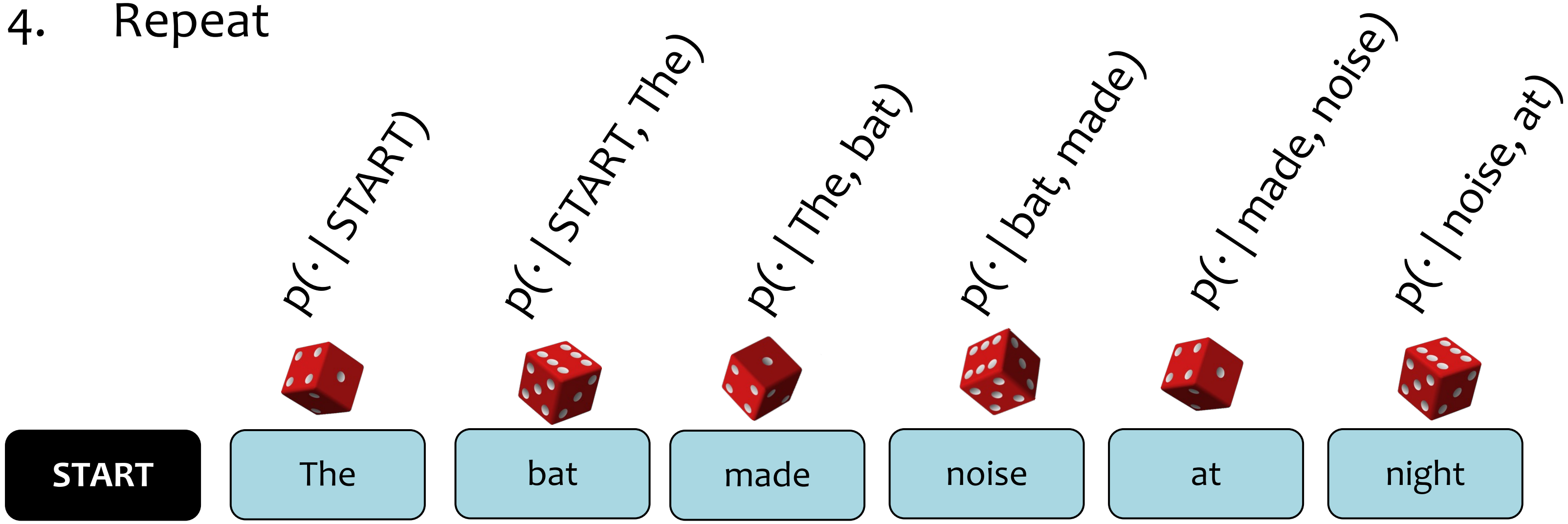
Sampling from a Language Model

Recall…

Question: How do we sample from a Language Model?

Answer:

1. Treat each probability distribution like a (50k-sided) weighted die
2. Pick the die corresponding to $p(w_t \mid w_{t-2}, w_{t-1})$
3. Roll that die and generate whichever word w_t lands face up
4. Repeat



Noisy Channel Models

- Prior to 2017, two tasks relied heavily on language models:
  - speech recognition
  - machine translation
- Definition: a **noisy channel model** combines a *transduction model* (probability of converting **y** to **x**) with a *language model* (probability of **y**)

$$\hat{\mathbf{y}} = \operatorname*{argmax}_{\mathbf{y}} p(\mathbf{y} \mid \mathbf{x}) = \operatorname*{argmax}_{\mathbf{y}} \underbrace{p(\mathbf{x} \mid \mathbf{y})}_{\text{transduction model}} \underbrace{p(\mathbf{y})}_{\text{language model}}$$

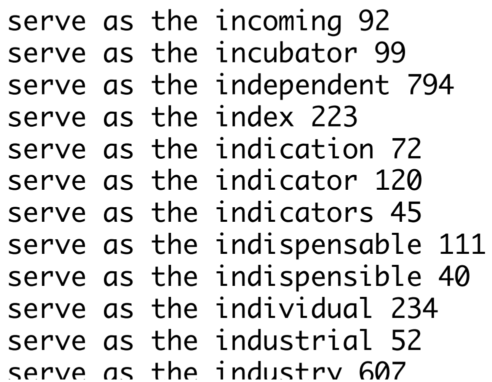
- **Goal:** to recover **y** from **x**
  - *For speech:* **x** is acoustic signal, **y** is transcription
  - *For machine translation:* **x** is sentence in source language, **y** is sentence in target language

Large (n-Gram) Language Models

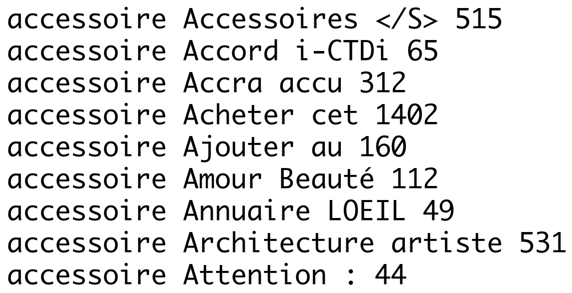
- The earliest (truly) large language models were n-gram models
- Google n-Grams:
  - 2006: first release, English n-grams
    - trained on **1 trillion tokens** of web text (95 billion sentences)
    - included 1-grams, 2-grams, 3-grams, 4-grams, and 5-grams
  - 2009 – 2010: n-grams in Japanese, Chinese, Swedish, Spanish, Romanian, Portuguese, Polish, Dutch, Italian, French, German, Czech

English n-gram model is ~**3 billion parameters**

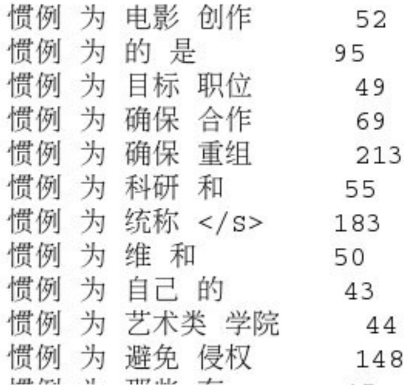
Number of unigrams:	13,588,391
Number of bigrams:	314,843,401
Number of trigrams:	977,069,902
Number of fourgrams:	1,313,818,354
Number of fivegrams:	1,176,470,663



```
serve as the incoming 92
serve as the incubator 99
serve as the independent 794
serve as the index 223
serve as the indication 72
serve as the indicator 120
serve as the indicators 45
serve as the indispensable 111
serve as the indispensible 40
serve as the individual 234
serve as the industrial 52
serve as the industry 607
```



```
accessoire Accessoires </S> 515
accessoire Accord i-CTDi 65
accessoire Accra accu 312
accessoire Acheter cet 1402
accessoire Ajouter au 160
accessoire Amour Beauté 112
accessoire Annuaire LOEIL 49
accessoire Architecture artiste 531
accessoire Attention : 44
```



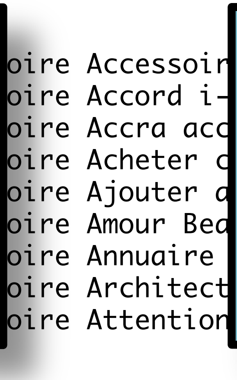
```
惯例 为 电影 创作      52
惯例 为 的 是         95
惯例 为 目标 职位      49
惯例 为 确保 合作      69
惯例 为 确保 重组      213
惯例 为 科研 和       55
惯例 为 统称 </s>    183
惯例 为 维 和         50
惯例 为 自己 的       43
惯例 为 艺术类 学院     44
惯例 为 避免 侵权      148
```

Large (n-Gram) Language Models

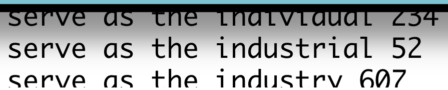
- The earliest (truly) large language models were n-gram models
- Google n-Grams:
  - 2006: first release, English n-grams
    - trained on **1 trillion tokens** of web text (95 billion sentences)
    - included 1-grams, 2-grams, 3-grams, 4-grams, and 5-grams
  - 2009 – 2010: n-grams in Japanese, Chinese, Swedish, Spanish, Romanian, Portuguese, Polish, Dutch, Italian, French, German, Czech

English n-gram model is ~**3 billion parameters**

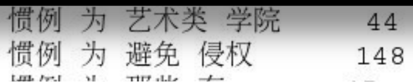
Number of unigrams:	13,588,391
Number of bigrams:	314,843,401
Number of trigrams:	977,069,902
Number of fourgrams:	1,313,818,354
Number of fivegrams:	1,176,470,663



```
oire Accessoir
oire Accord i-
oire Accra acc
oire Acheter c
oire Ajouter a
oire Amour Bea
oire Annuaire
oire Architect
oire Attention
```



```
serve as the individual 234
serve as the industrial 52
serve as the industry 607
```



```
惯例 为 艺术类 学院      44
惯例 为 避免 侵权       148
```

Q: Is this a large training set?

A: Yes!

Q: Is this a large model?

A: Yes!

How large are LLMs?

Comparison of some recent **large language models** (LLMs)

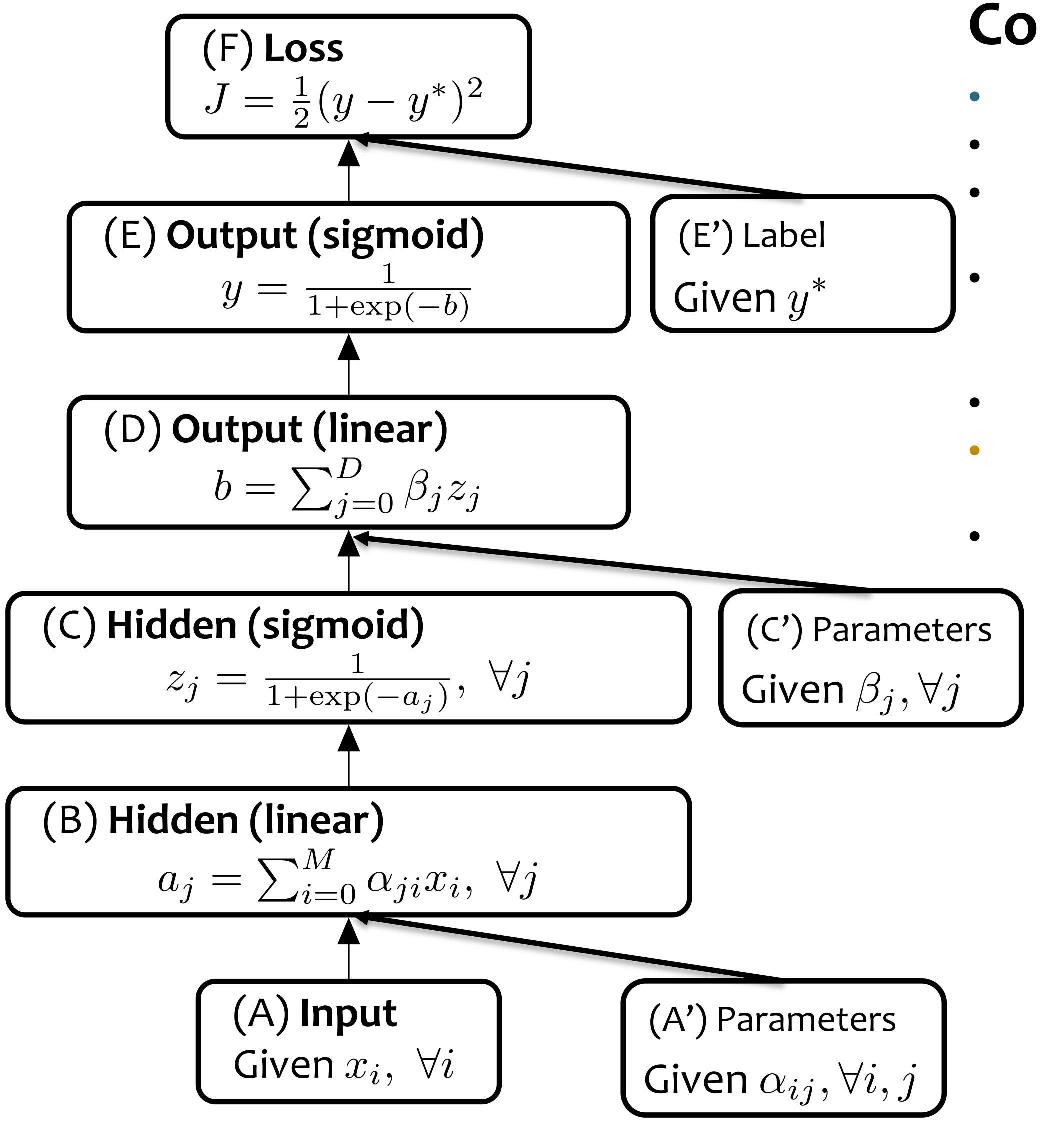
Model	Creators	Year of release	Training Data (# tokens)	Model Size (# parameters)
GPT-2	OpenAI	2019	~10 billion (40Gb)	1.5 billion
GPT-3 (cf. ChatGPT)	OpenAI	2020	300 billion	175 billion
PaLM	Google	2022	780 billion	540 billion
Chinchilla	DeepMind	2022	1.4 trillion	70 billion
LaMDA (cf. Bard)	Google	2022	1.56 trillion	137 billion
LLaMA	Meta	2023	1.4 trillion	65 billion
GPT-4	OpenAI	2023	?	?

Transformer Language Models

MODEL: GPT

Recall...

Ways of Drawing Neural Networks



(F) **Loss**
$J = \frac{1}{2}(y - y^*)^2$

(E) **Output (sigmoid)**
$y = \frac{1}{1+\exp(-b)}$

(E') Label
Given y^*

(D) **Output (linear)**
$b = \sum_{j=0}^{D} \beta_j z_j$

(C) **Hidden (sigmoid)**
$z_j = \frac{1}{1+\exp(-a_j)}, \ \forall j$

(C') Parameters
Given $\beta_j, \forall j$

(B) **Hidden (linear)**
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \ \forall j$

(A) **Input**
Given $x_i, \ \forall i$

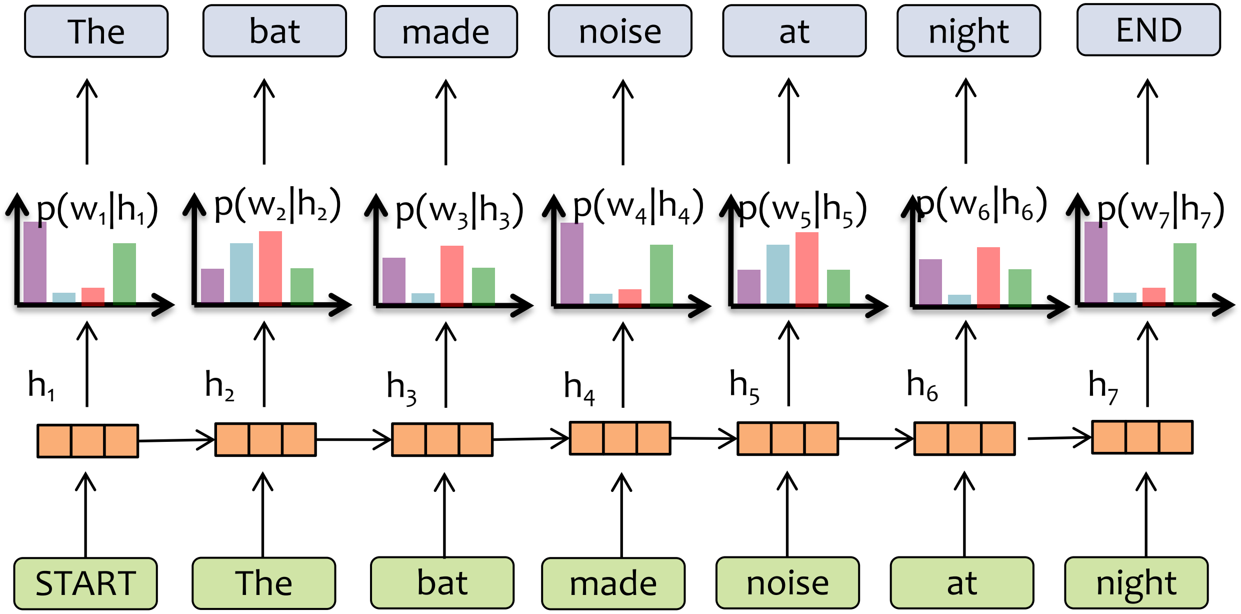
(A') Parameters
Given $\alpha_{ij}, \forall i, j$

Computation Graph

- The diagram represents an algorithm
- Nodes are **rectangles**
- One node per intermediate variable in the algorithm
- Node is labeled with the function that it computes (inside the box) and also the variable name (outside the box)
- *Edges are directed*
- Edges do not have labels (since they don't need them)
- For neural networks:
  - Each intercept term should appear as a node (if it's not folded in somewhere)
  - Each parameter should appear as a node
  - Each constant, e.g. a true label or a feature vector should appear in the graph
  - It's perfectly fine to include the loss

RNN Language Model

Recall...



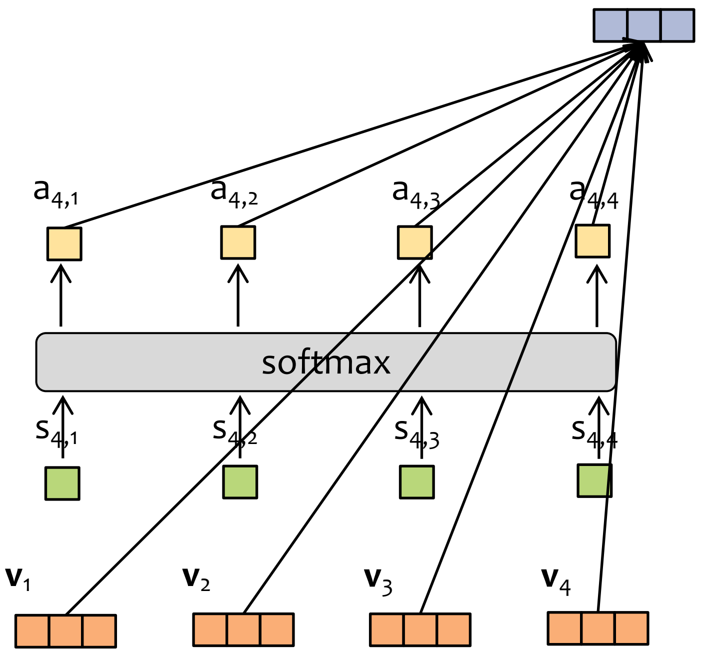
Key Idea:

(1) convert all previous words to a **fixed length vector**

(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \dots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \dots, w_1)$

RNNs and Forgetting

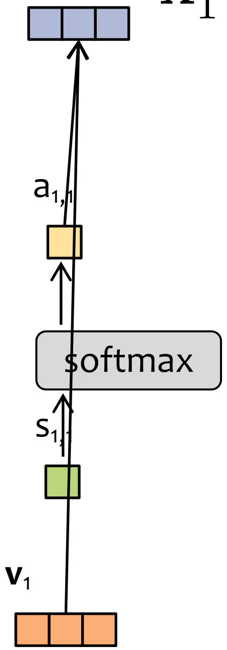
Attention



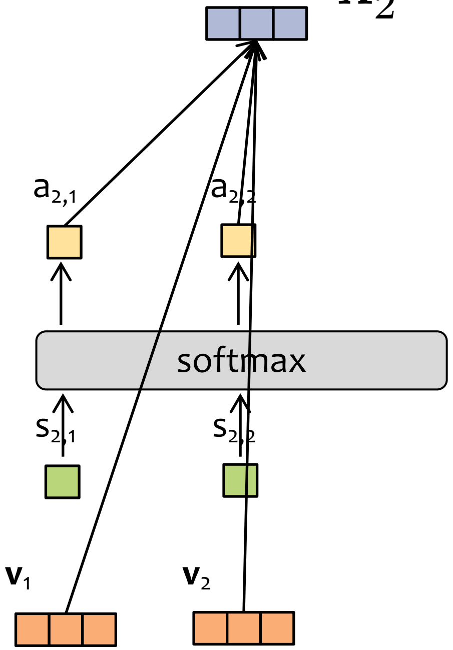
$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

Attention

$$\mathbf{x}'_1 = \sum_{j=1}^{1} a_{1,j}\mathbf{v}_j$$

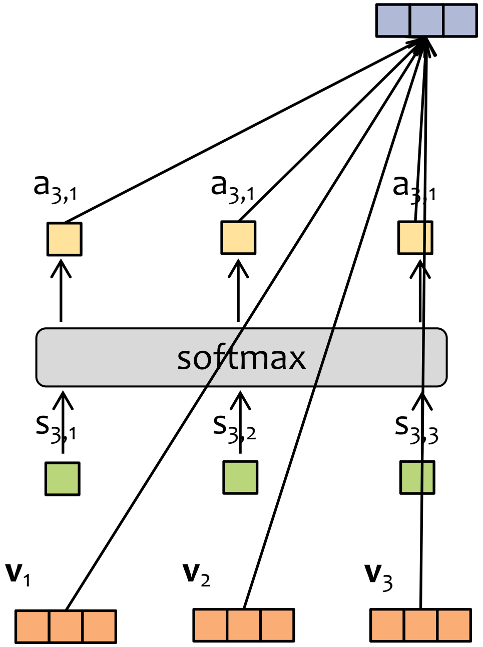


Attention



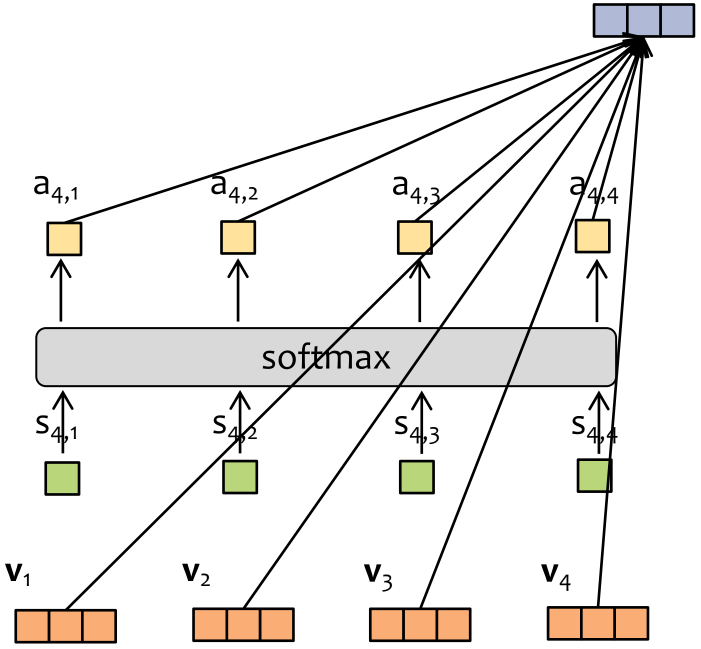
$$\mathbf{x}'_2 = \sum_{j=1}^{2} a_{2,j}\mathbf{v}_j$$

Attention



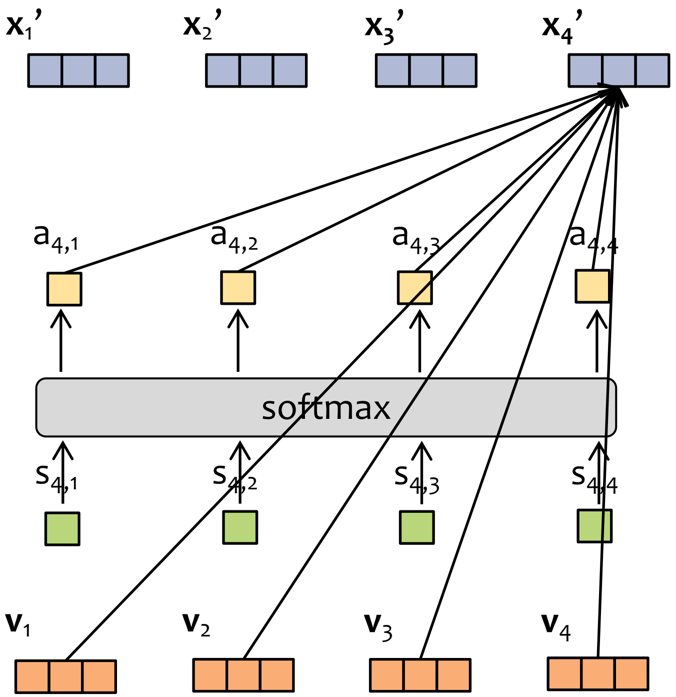
$$\mathbf{x}'_3 = \sum_{j=1}^{3} a_{3,j}\mathbf{v}_j$$

Attention



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

Attention



$$\mathbf{x}'_t = \sum_{j=1}^{t} a_{t,j} \mathbf{v}_j$$

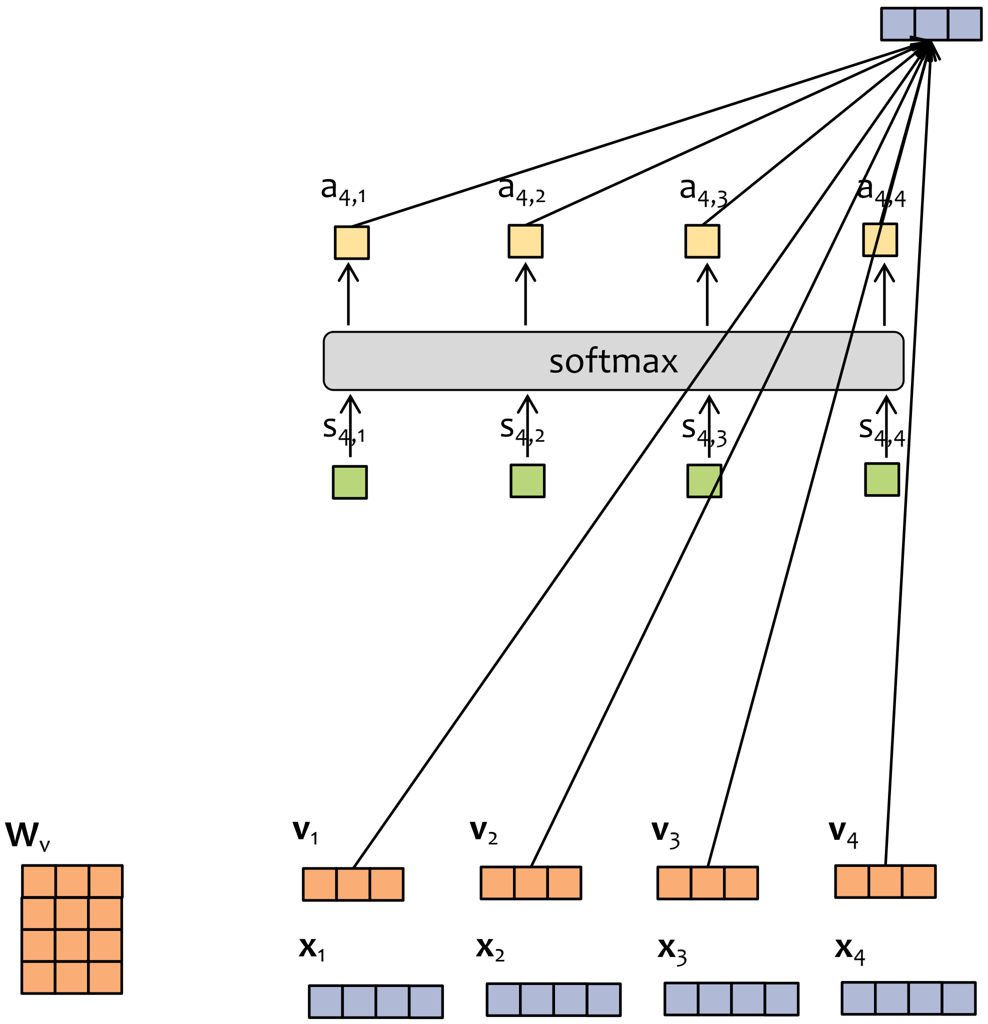
attention weights

scores

values

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

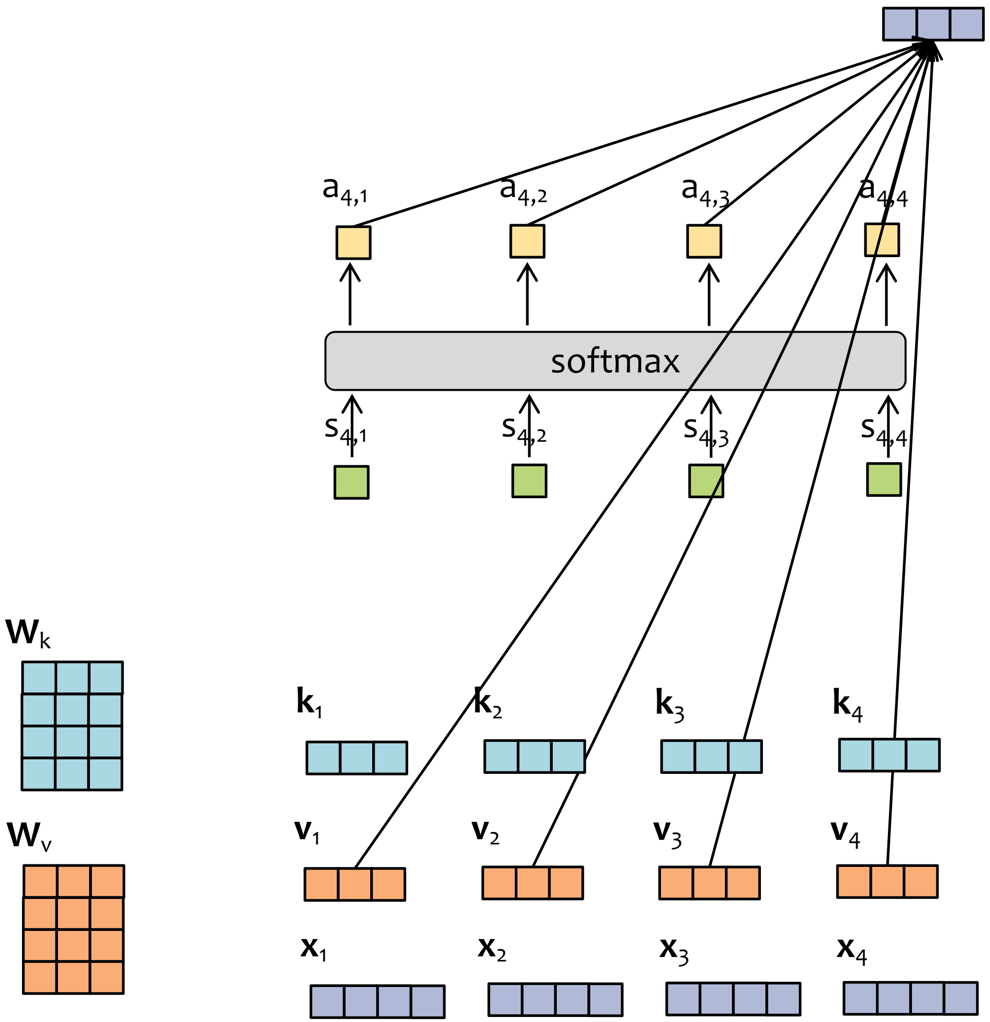


$$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$$

values

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

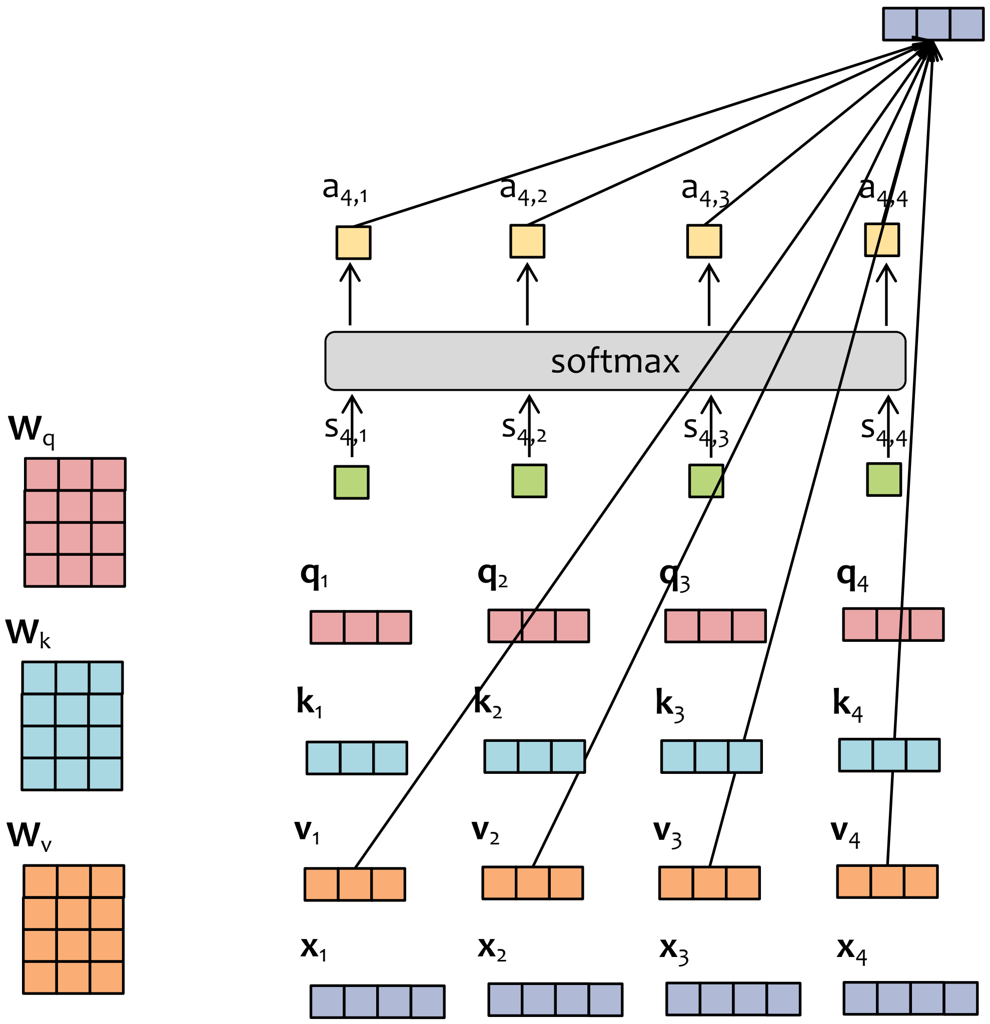


$$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$$ keys

$$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$$ values

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$



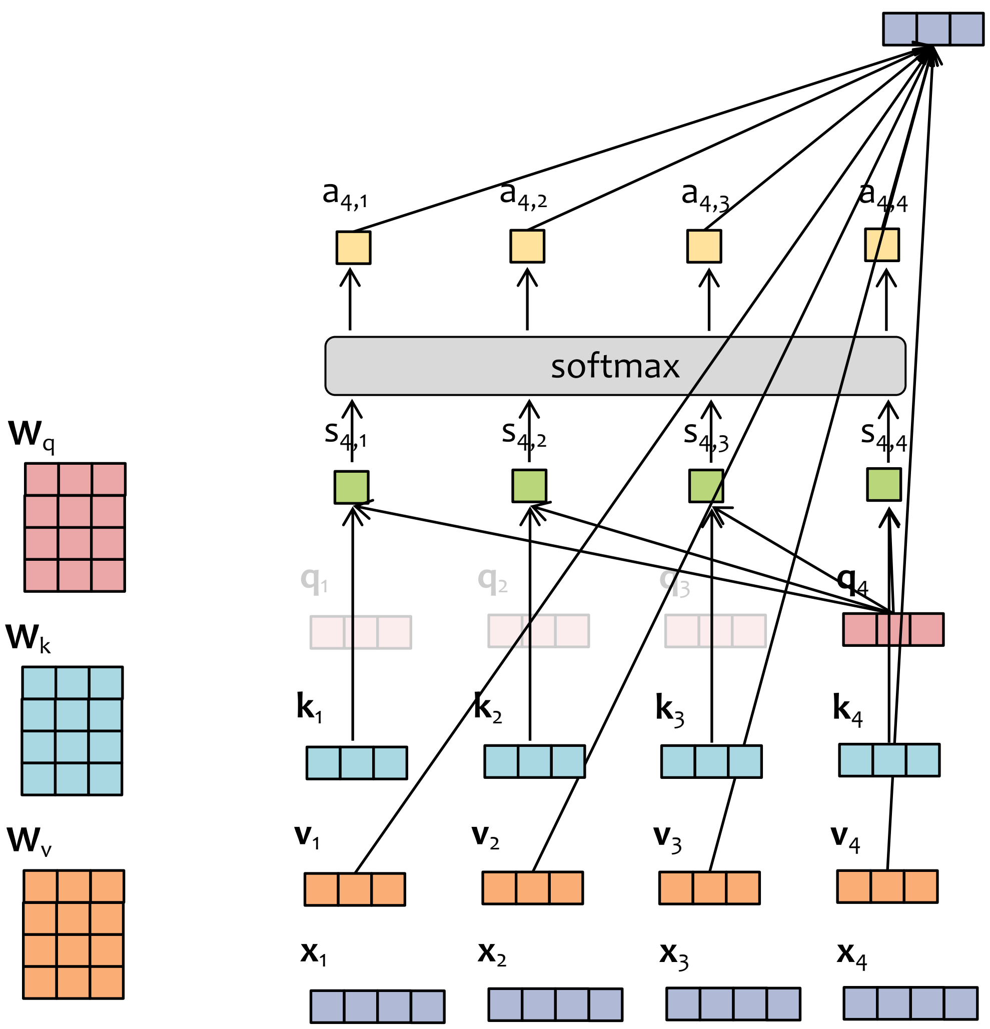
$\mathbf{q}_j = \mathbf{W}_q^T\mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T\mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T\mathbf{x}_j$ values

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$



$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

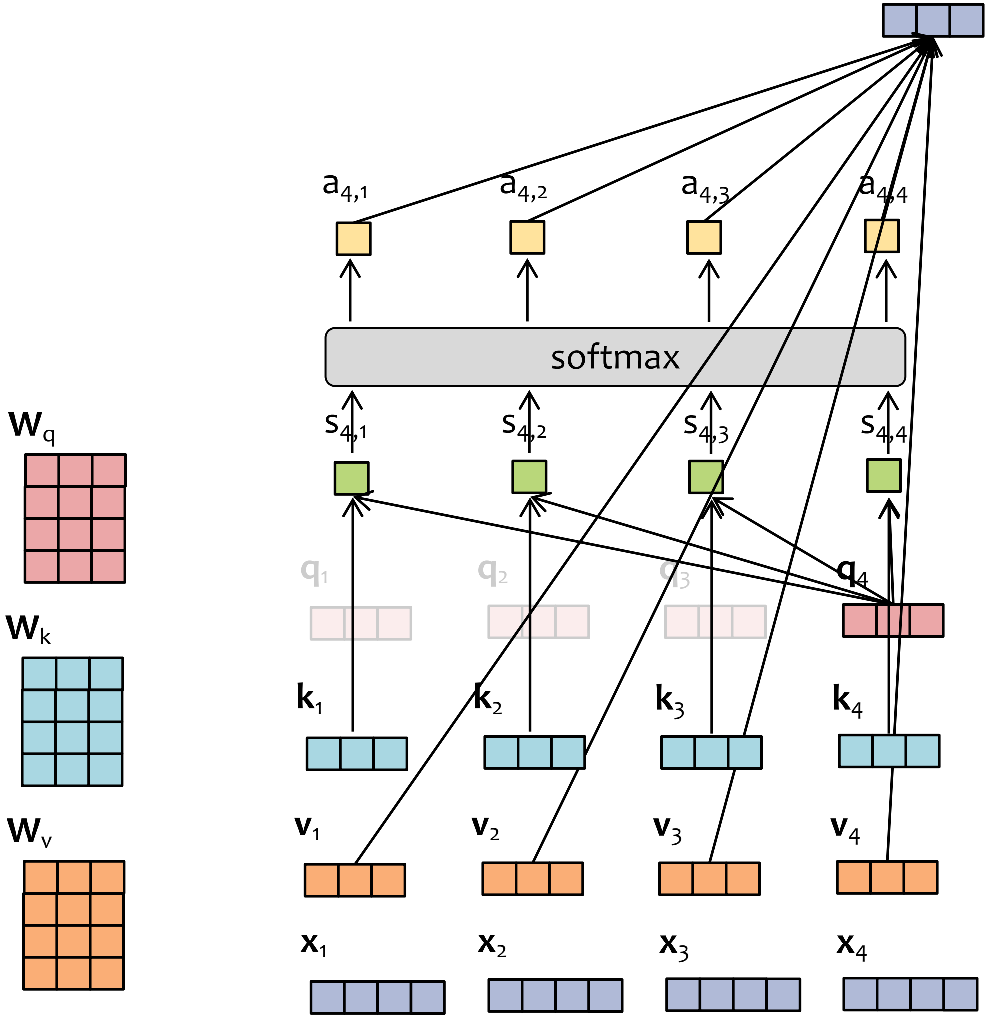
$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$



$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$attention weights

$s_{4,j} = \mathbf{k}_j^T\mathbf{q}_4/\sqrt{d_k}$scores

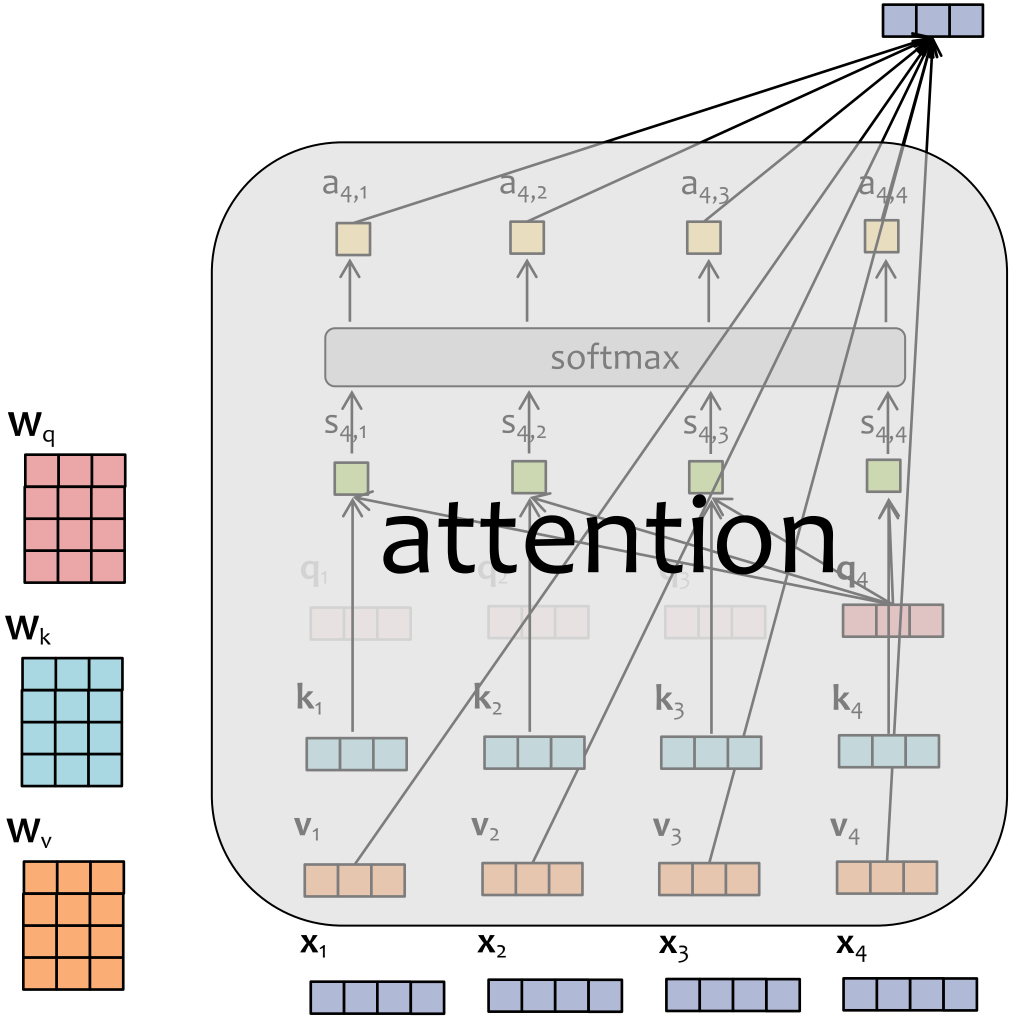
$\mathbf{q}_j = \mathbf{W}_q^T\mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T\mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T\mathbf{x}_j$ values

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$



$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$ attention weights

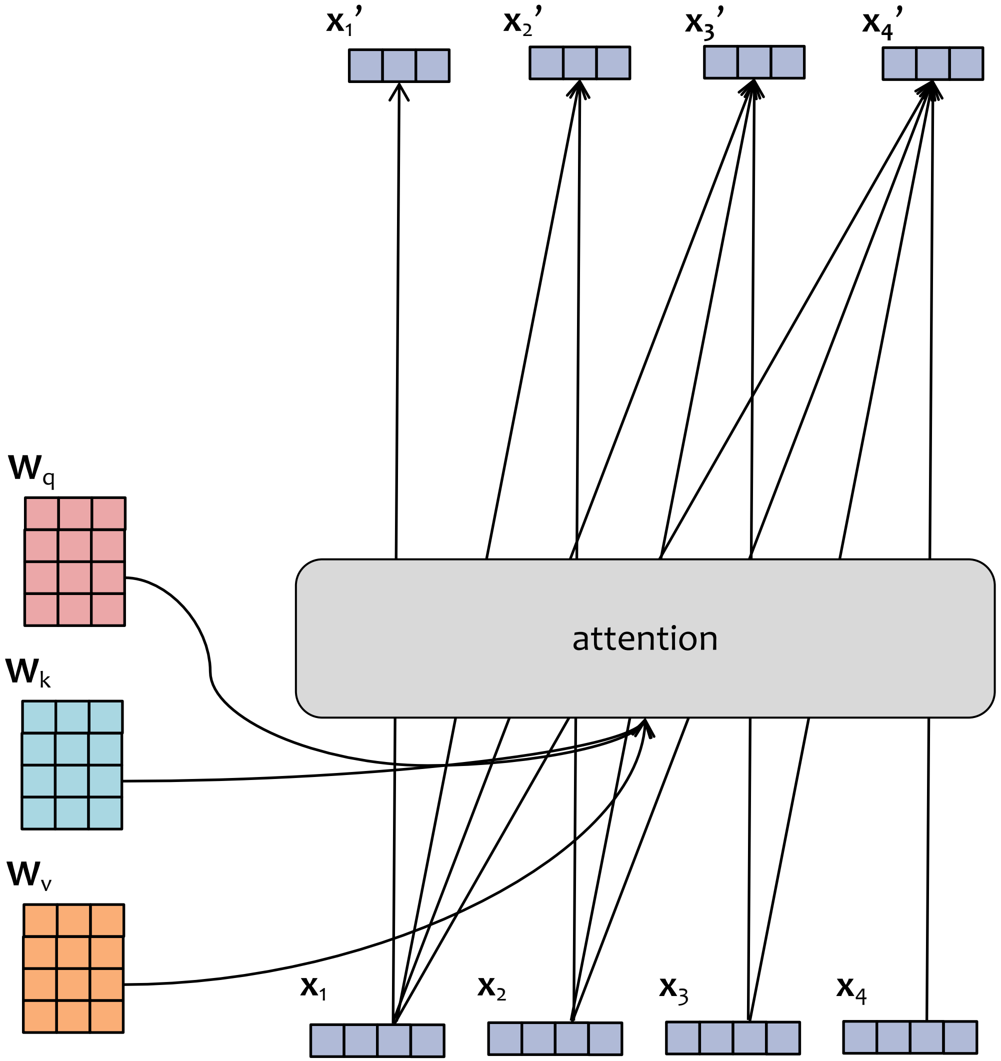
$s_{4,j} = \mathbf{k}_j^T\mathbf{q}_4/\sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T\mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T\mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T\mathbf{x}_j$ values

Scaled Dot-Product Attention



$$\mathbf{x}'_t = \sum_{j=1}^{t} a_{t,j}\mathbf{v}_j$$

$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t)$ attention weights

$s_{t,j} = \mathbf{k}_j^T \mathbf{q}_t / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

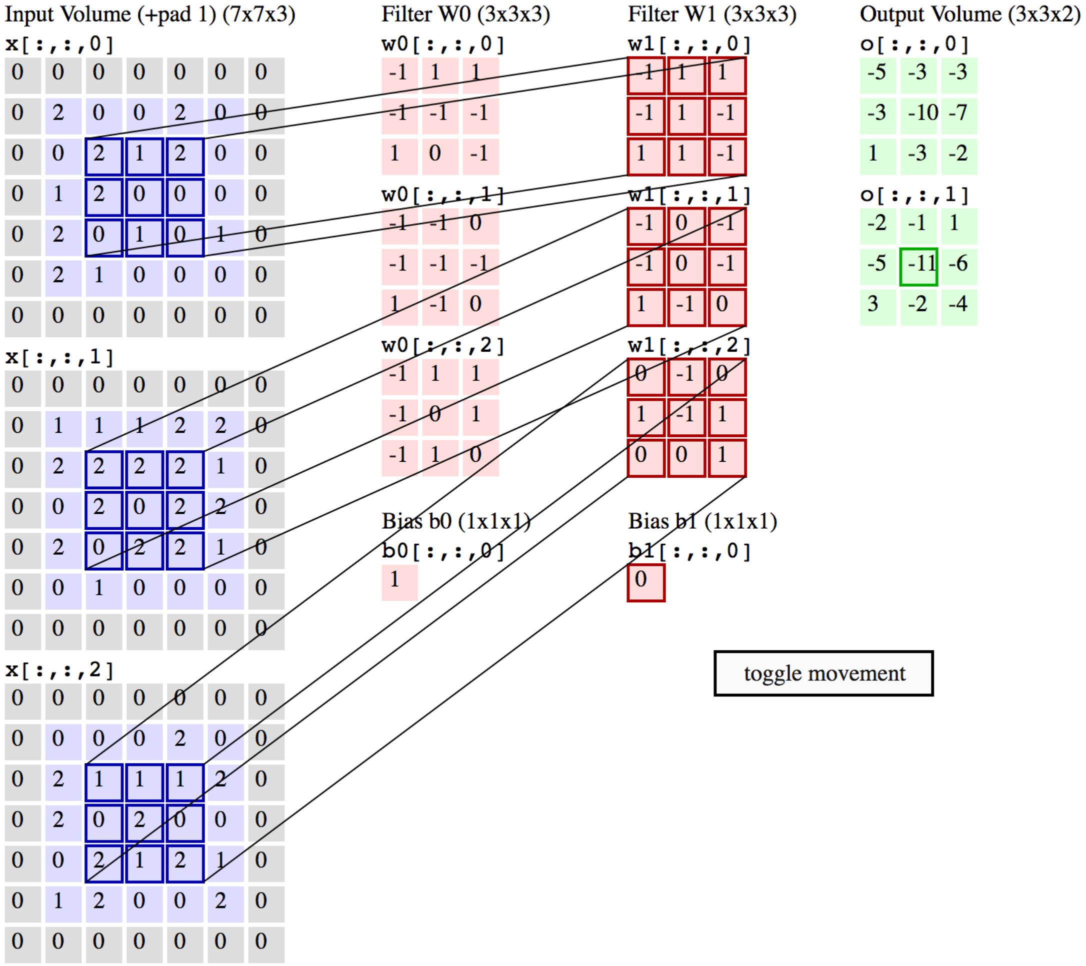
$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Animation of 3D Convolution

Recall...

http://cs231n.github.io/convolutional-networks/



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	2	0	0	2	0	0
0	0	2	1	2	0	0
0	1	2	0	0	0	0
0	2	0	1	0	1	0
0	2	1	0	0	0	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	1	1	1	2	2	0
0	2	2	2	2	1	0
0	0	2	0	2	2	0
0	2	0	2	2	1	0
0	0	1	0	0	0	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	0	0	0	2	0	0
0	2	1	1	1	2	0
0	2	0	2	0	0	0
0	0	2	1	2	1	0
0	1	2	0	0	2	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

-1	1	1
-1	-1	-1
1	0	-1

w0[:,:,1]

-1	-1	0
-1	-1	-1
1	-1	0

w0[:,:,2]

-1	1	1
-1	0	1
-1	1	0

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

-1	1	1
-1	1	-1
1	1	-1

w1[:,:,1]

-1	0	-1
-1	0	-1
1	-1	0

w1[:,:,2]

0	-1	0
1	-1	1
0	0	1

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

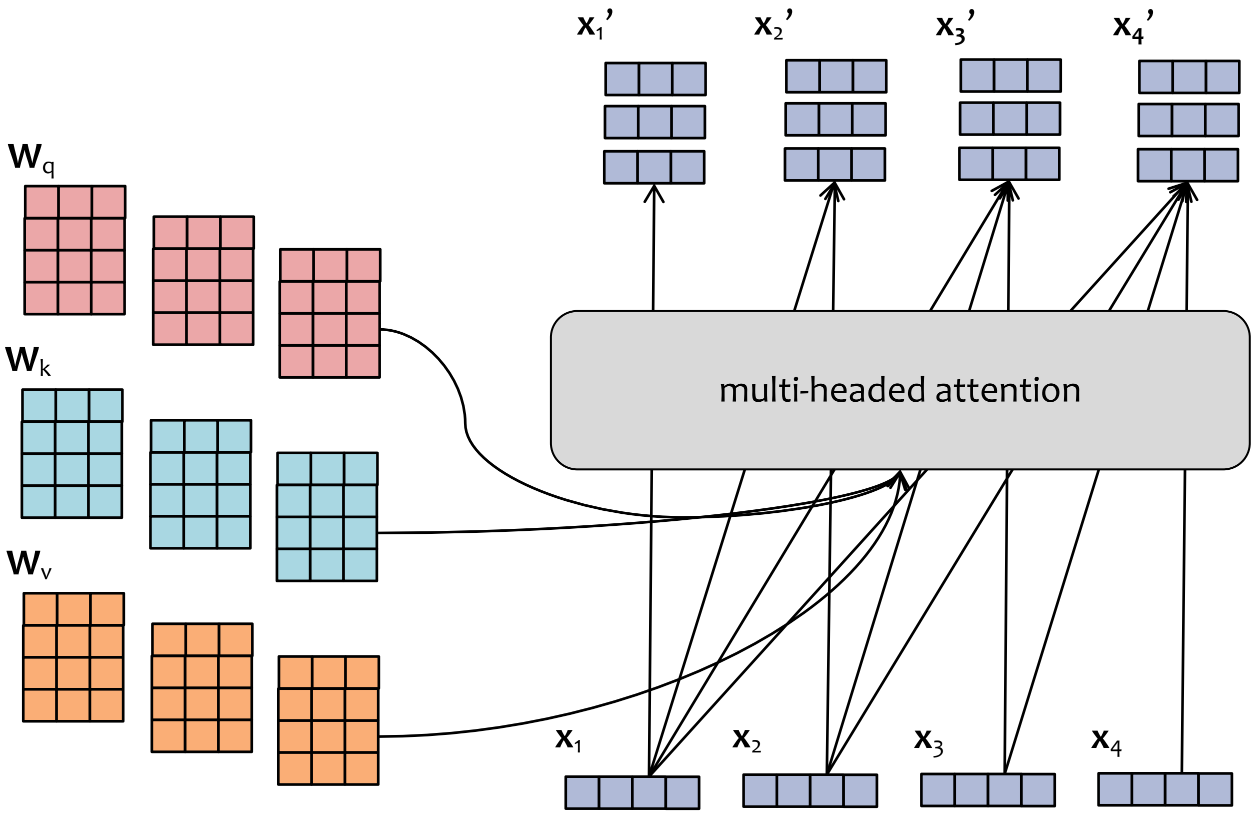
-5	-3	-3
-3	-10	-7
1	-3	-2

o[:,:,1]

-2	-1	1
-5	-11	-6
3	-2	-4

Figure from Fei-Fei Li & Andrej Karpathy & Justin Johnson (CS231N)

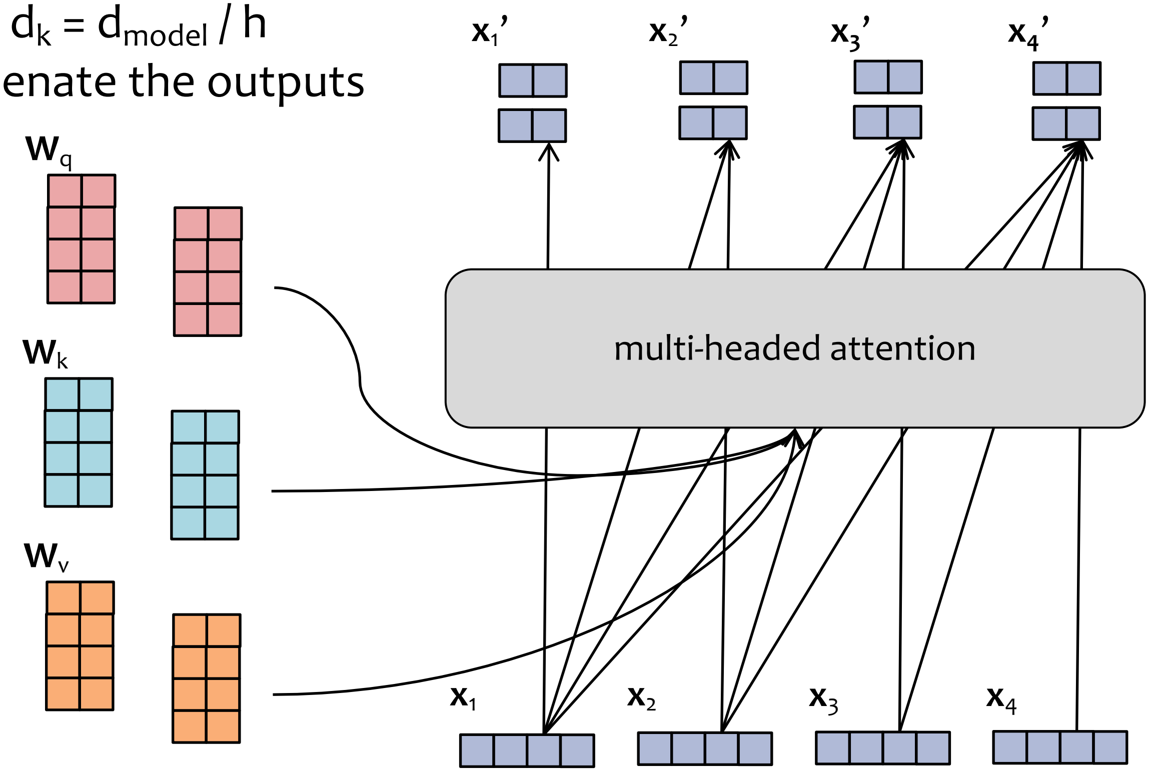
Multi-headed Attention



- Just as we can have **multiple channels** in a **convolution** layer, we can use **multiple heads** in an **attention** layer
- Each head gets **its own parameters**
- We can **concatenate** all the outputs to get a single vector for each time step

Multi-headed Attention

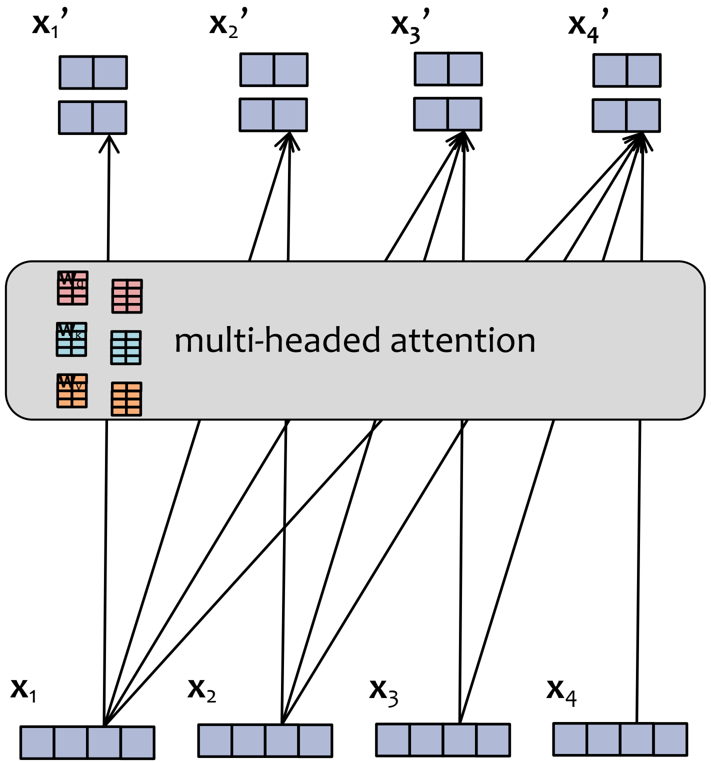
- To ensure the dimension of the **input** embedding $\mathbf{x}_t$ is the same as the **output** embedding $\mathbf{x}_t'$, Transformers usually choose the embedding sizes and number of heads appropriately:
  - d_{model} = dim. of inputs
  - d_k = dim. of each output
  - h = # of heads
  - Choose $d_k = d_{model} / h$
- Then concatenate the outputs



- Just as we can have **multiple channels** in a **convolution** layer, we can use **multiple heads** in an **attention** layer
- Each head gets **its own parameters**
- We can **concatenate** all the outputs to get a single vector for each time step

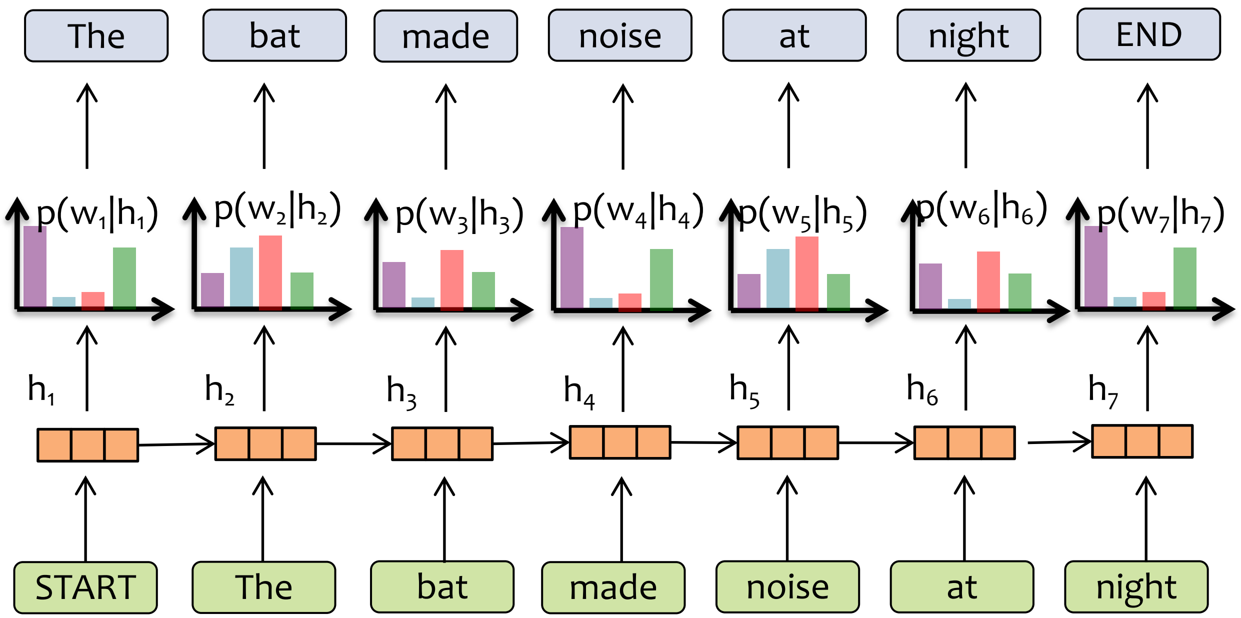
Multi-headed Attention

- To ensure the dimension of the **input** embedding $\mathbf{x}_t$ is the same as the **output** embedding $\mathbf{x}_t$', Transformers usually choose the embedding sizes and number of heads appropriately:
  - d_{model} = dim. of inputs
  - d_k = dim. of each output
  - h = # of heads
  - Choose $d_k = d_{model} / h$
- Then concatenate the outputs



- Just as we can have **multiple channels** in a **convolution** layer, we can use **multiple heads** in an **attention** layer
- Each head gets **its own parameters**
- We can **concatenate** all the outputs to get a single vector for each time step

RNN Language Model



<u>Key Idea</u>:

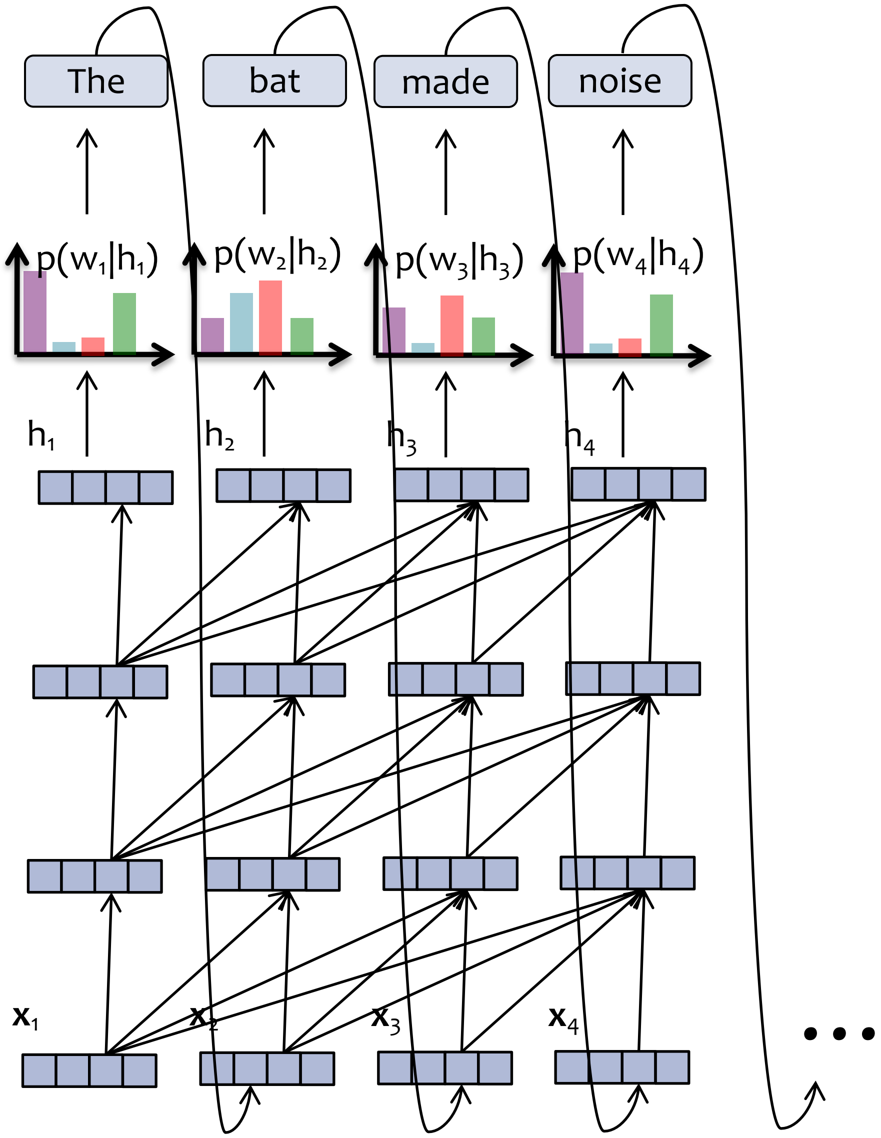
(1) convert all previous words to a **fixed length vector**

(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

Transformer Language Model

Important!

- RNN computation graph grows **linearly** with the number of input tokens
- Transformer-LM computation graph grows **quadratically** with the number of input tokens



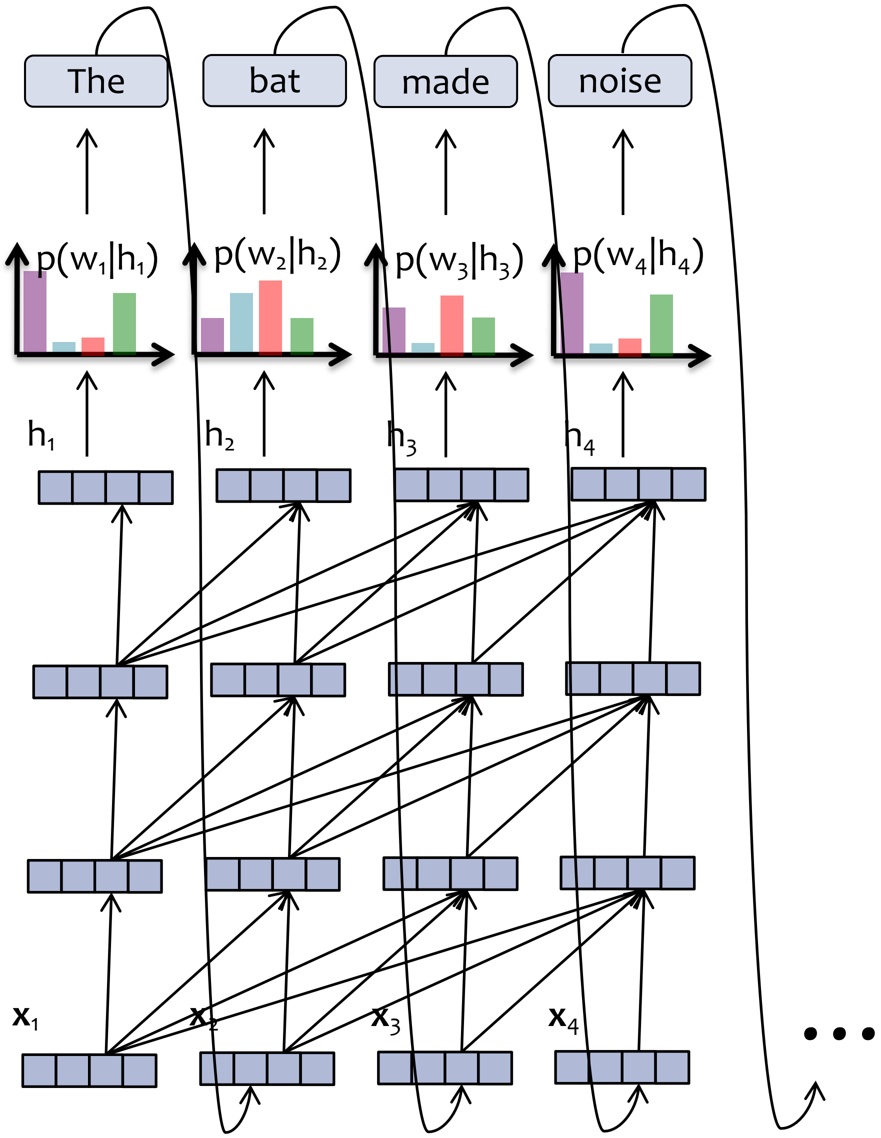
Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer.**

The language model part is just like an RNN-LM!

Transformer Language Model

Important!

- RNN computation graph grows **linearly** with the number of input tokens
- Transformer-LM computation graph grows **quadratically** with the number of input tokens



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer.**

The language model part is just like an RNN-LM!

Layer Normalization

- *The Problem*: **internal covariate shift** occurs during training of a deep network when a small change in the low layers amplifies into a large change in the high layers
- *One Solution:* **Layer normalization** normalizes each layer and learns elementwise gain/bias
- Such normalization allows for higher learning rates (for **faster convergence**) without issues of diverging gradients

Given input $\mathbf{a} \in \mathbb{R}^K$, LayerNorm computes output $\mathbf{b} \in \mathbb{R}^K$:

$$\mathbf{b} = \boldsymbol{\gamma} \odot \frac{\mathbf{a} - \mu}{\sigma} \oplus \boldsymbol{\beta}$$

where we have mean $\mu = \frac{1}{K} \sum_{k=1}^{K} a_k$,
standard deviation $\sigma = \sqrt{\frac{1}{K} \sum_{k=1}^{K} (a_k - \mu)^2}$,
and parameters $\boldsymbol{\gamma} \in \mathbb{R}^K$, $\boldsymbol{\beta} \in \mathbb{R}^K$.
$\odot$ and $\oplus$ denote elementwise multiplication and addition.

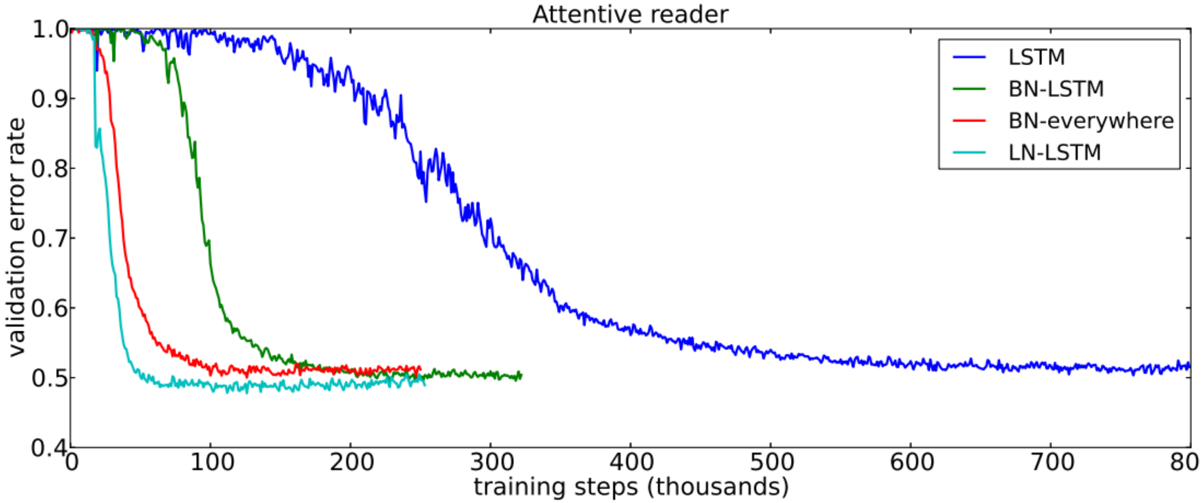
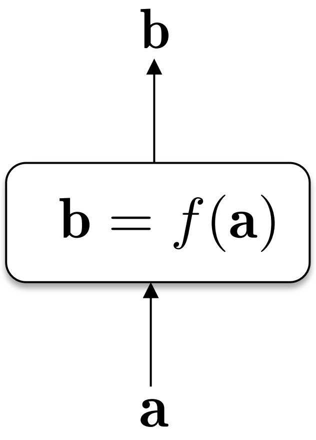


Figure from https://arxiv.org/pdf/1607.06450.pdf

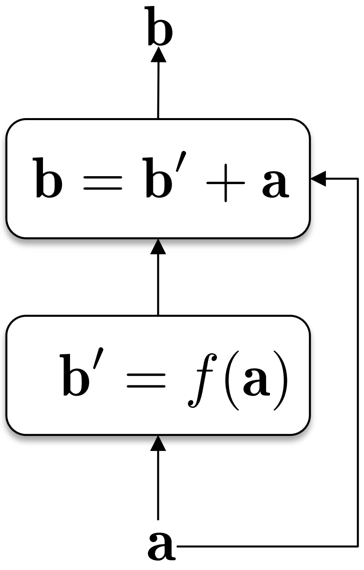
Residual Connections

- *The Problem:* as network depth grows very large, a **performance degradation** occurs that is not explained by overfitting (i.e. train / test error both worsen)
- *One Solution:* **Residual connections** pass a copy of the input alongsidethe
- These residual connections allow for **effective training of very deep networks** that perform better than their shallower (though still deep) counterparts

Plain Connection



Residual Connection



$\mathbf{b}$
$\mathbf{b} = \mathbf{b}' + \mathbf{a}$
$\mathbf{b}' = f(\mathbf{a})$
$\mathbf{a}$

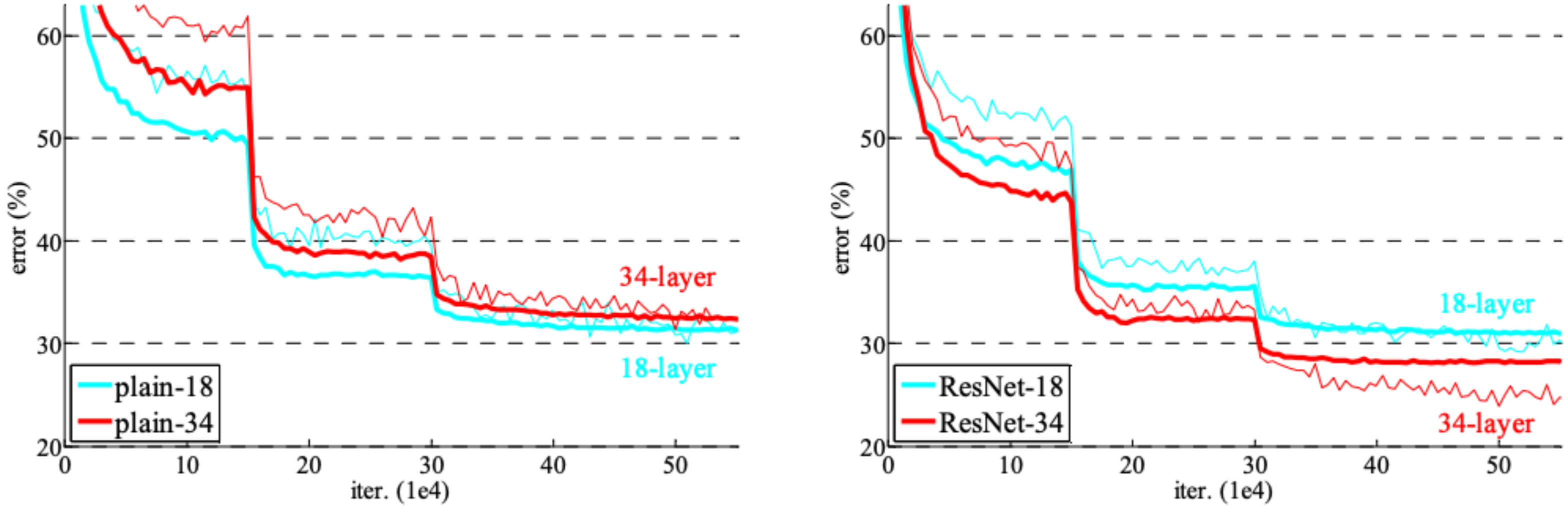
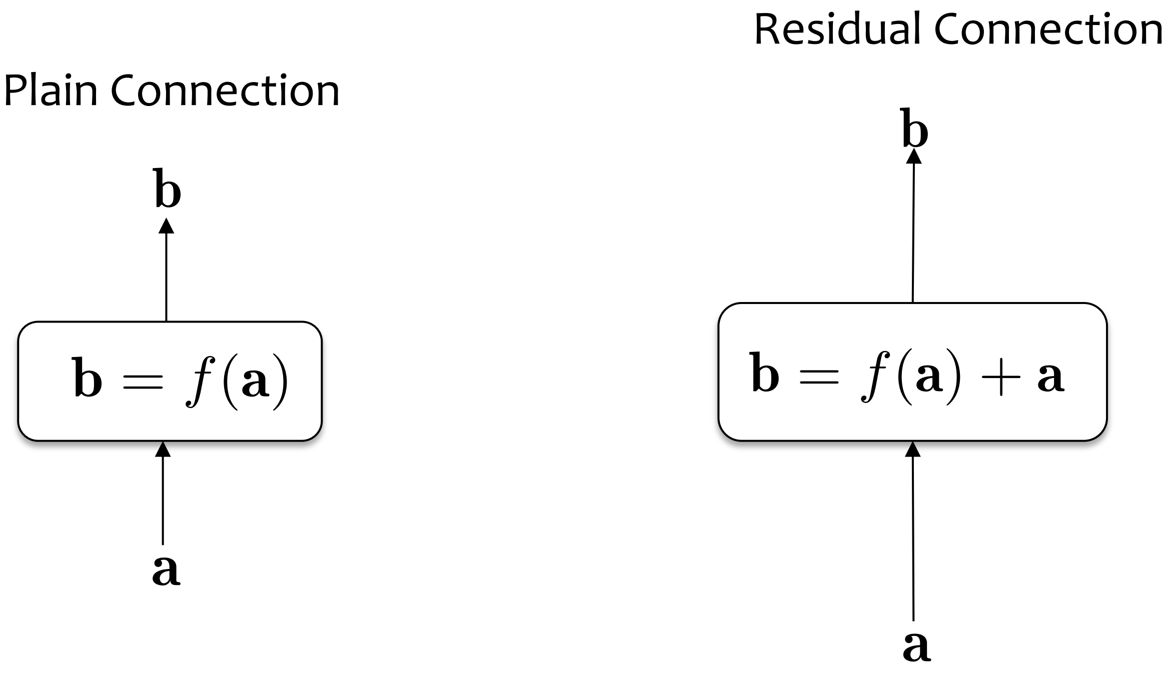


Figure from https://arxiv.org/pdf/1512.03385.pdf

Residual Connections

- *The Problem:* as network depth grows very large, a **performance degradation** occurs that is not explained by overfitting (i.e. train / test error both worsen)
- *One Solution:* **Residual connections** pass a copy of the input alongsidethe
- These residual connections allow for **effective training of very deep networks** that perform better than their shallower (though still deep) counterparts



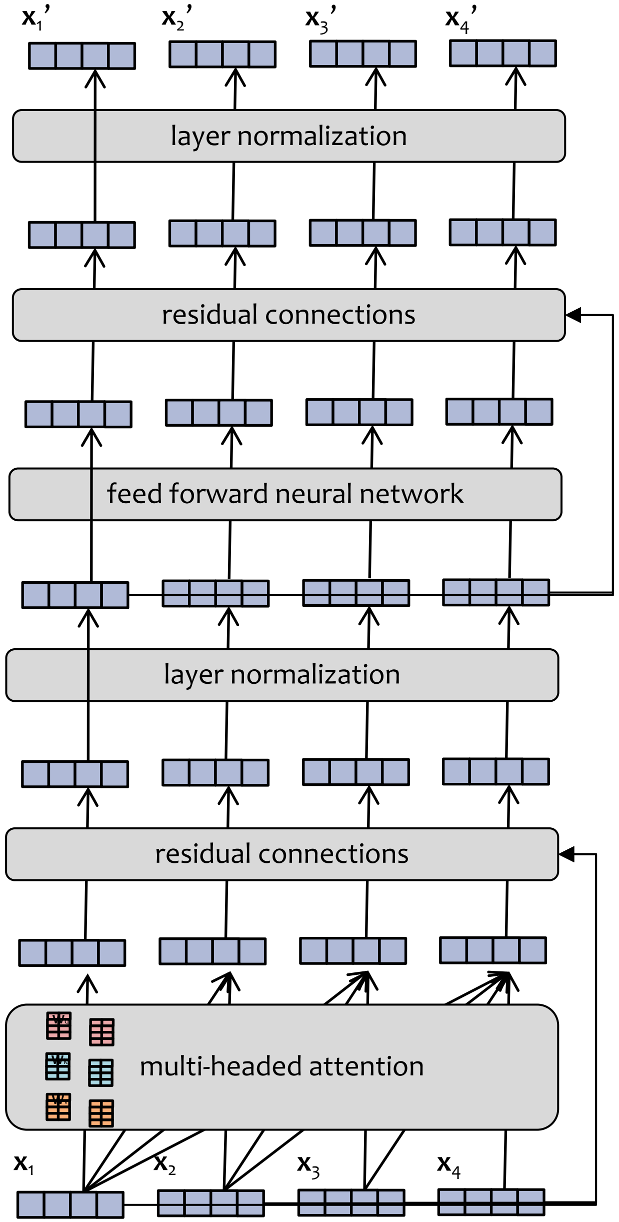
Plain Connection
b
$\mathbf{b} = f(\mathbf{a})$
a
Residual Connection
b
$\mathbf{b} = f(\mathbf{a}) + \mathbf{a}$
a

Why are residual connections helpful?

Instead of f(a) having to learn a full transformation of a, f(a) only needs to learn an additive modification of a (i.e. the residual).

Figure from https://arxiv.org/pdf/1512.03385.pdf

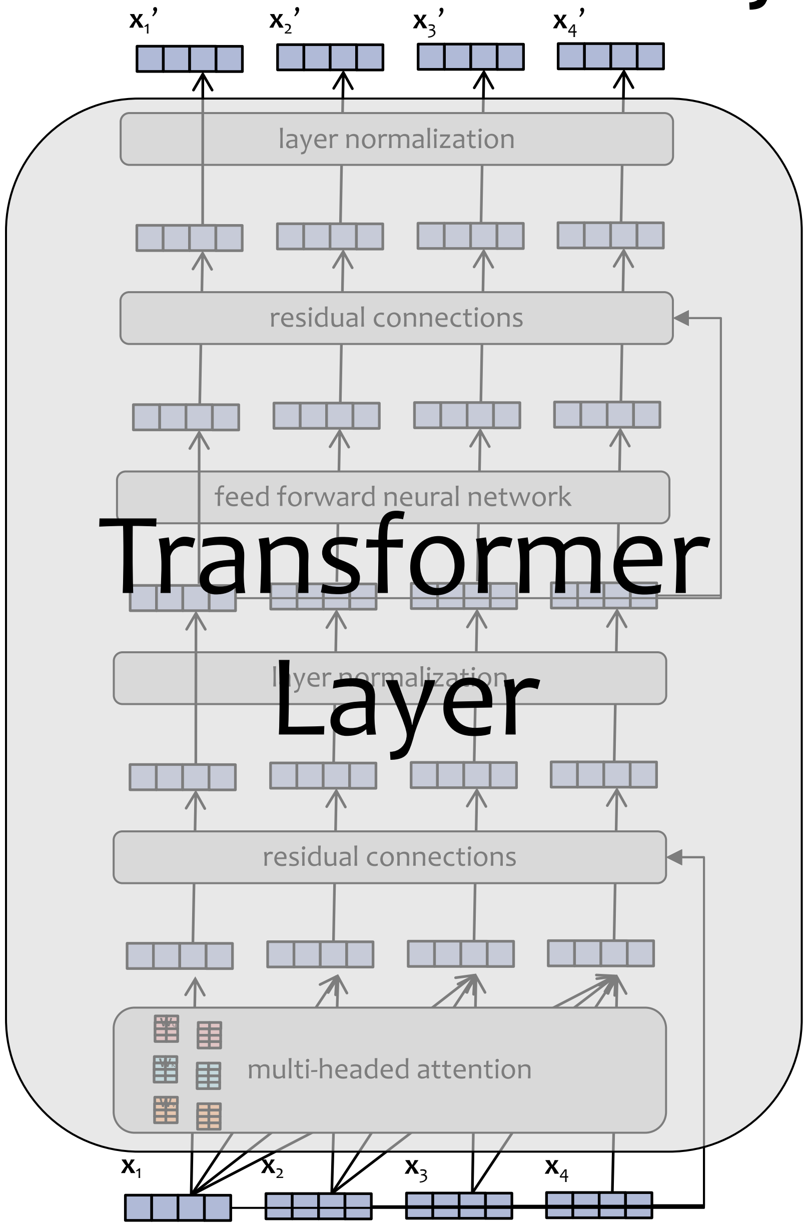
Transformer Layer



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

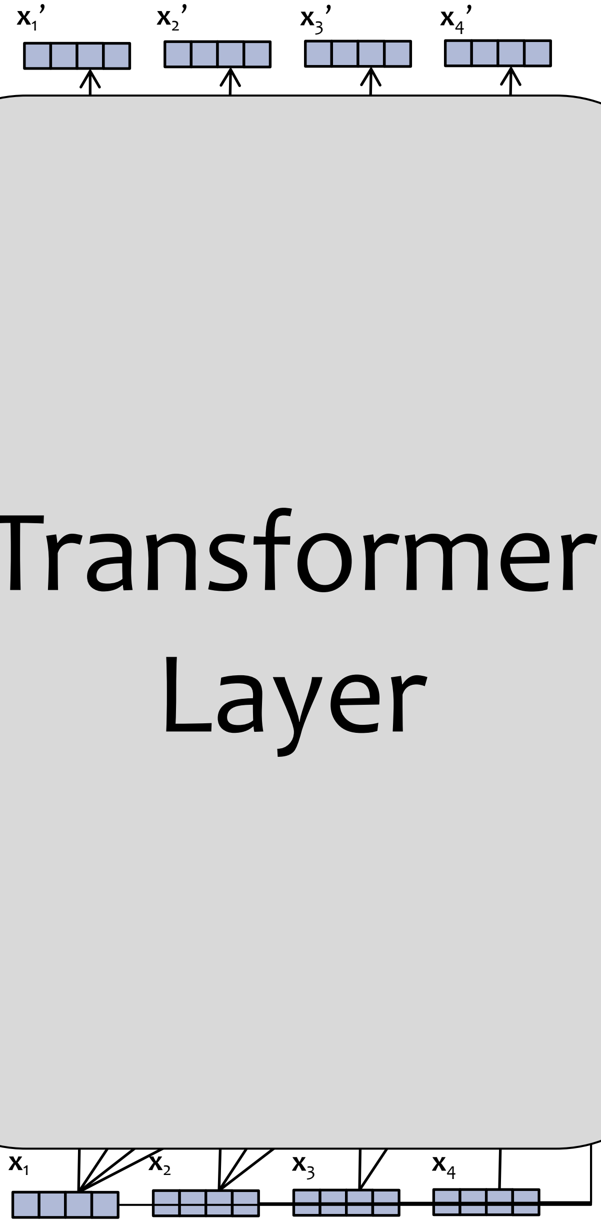
Transformer Layer



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Transformer Layer



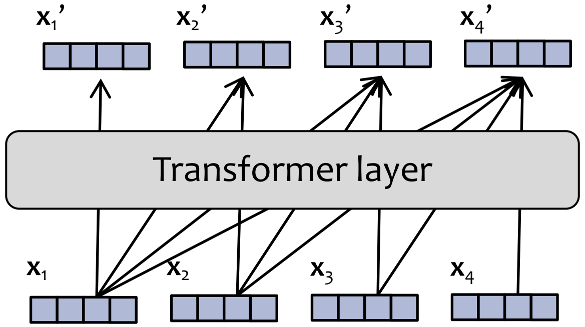
Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

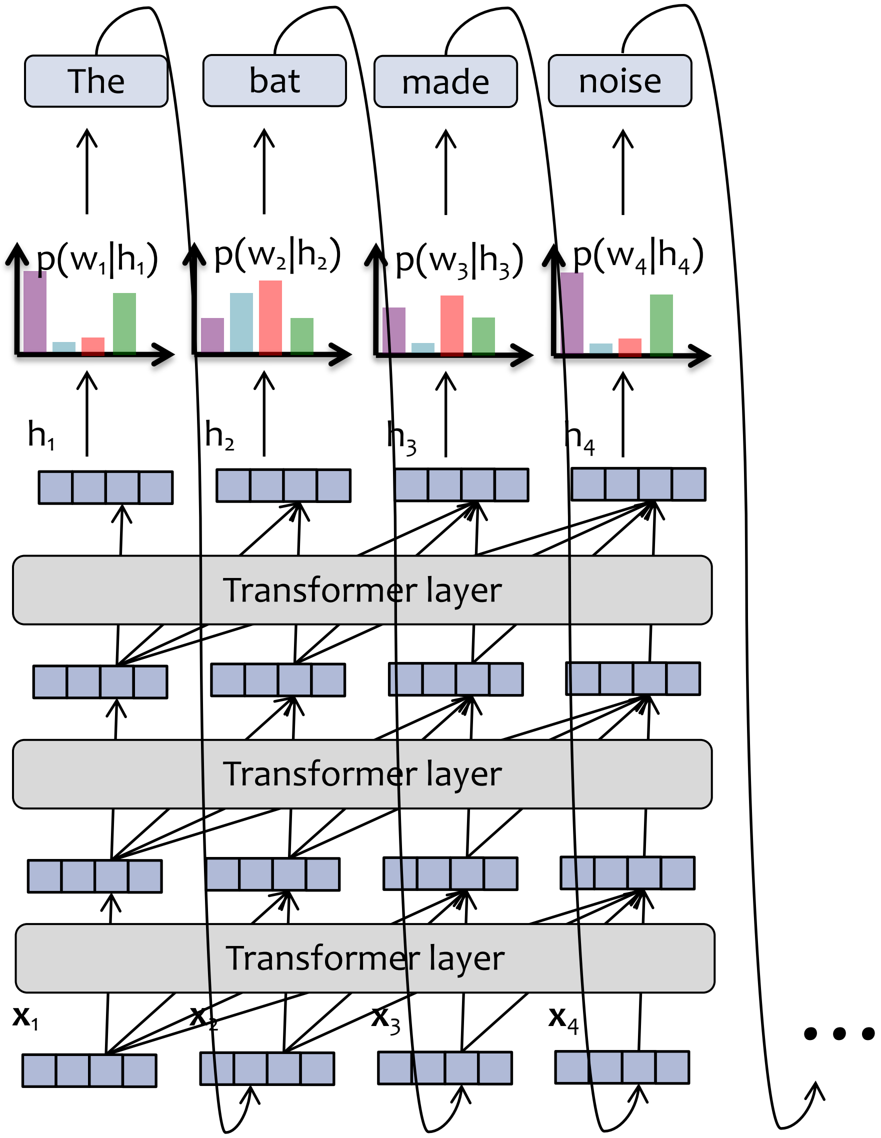
Transformer Layer

Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections



Transformer Language Model



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer.**

The language model part is just like an RNN-LM.

In-Class Poll

Question:

Suppose we have the following input embeddings and attention weights:

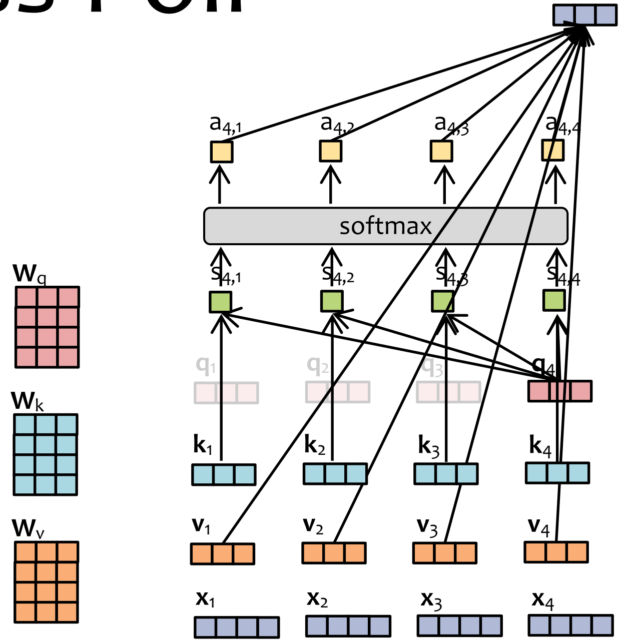
- $x_1 = [1,0,0,0]$ $a_{4,1} = 0.1$
- $x_2 = [0,1,0,0]$ $a_{4,2} = 0.2$
- $x_3 = [0,0,2,0]$ $a_{4,3} = 0.6$
- $x_4 = [0,0,0,1]$ $a_{4,4} = 0.1$

And $W_v = I$. Then we can compute x_4'.

Now suppose we swap the embeddings x_2 and x_3 such that

- $x_2 = [0,0,2,0]$
- $x_3 = [0,1,0,0]$

What is the new value of x_4'?



$$\mathbf{x}_4' = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$ attention weights

$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

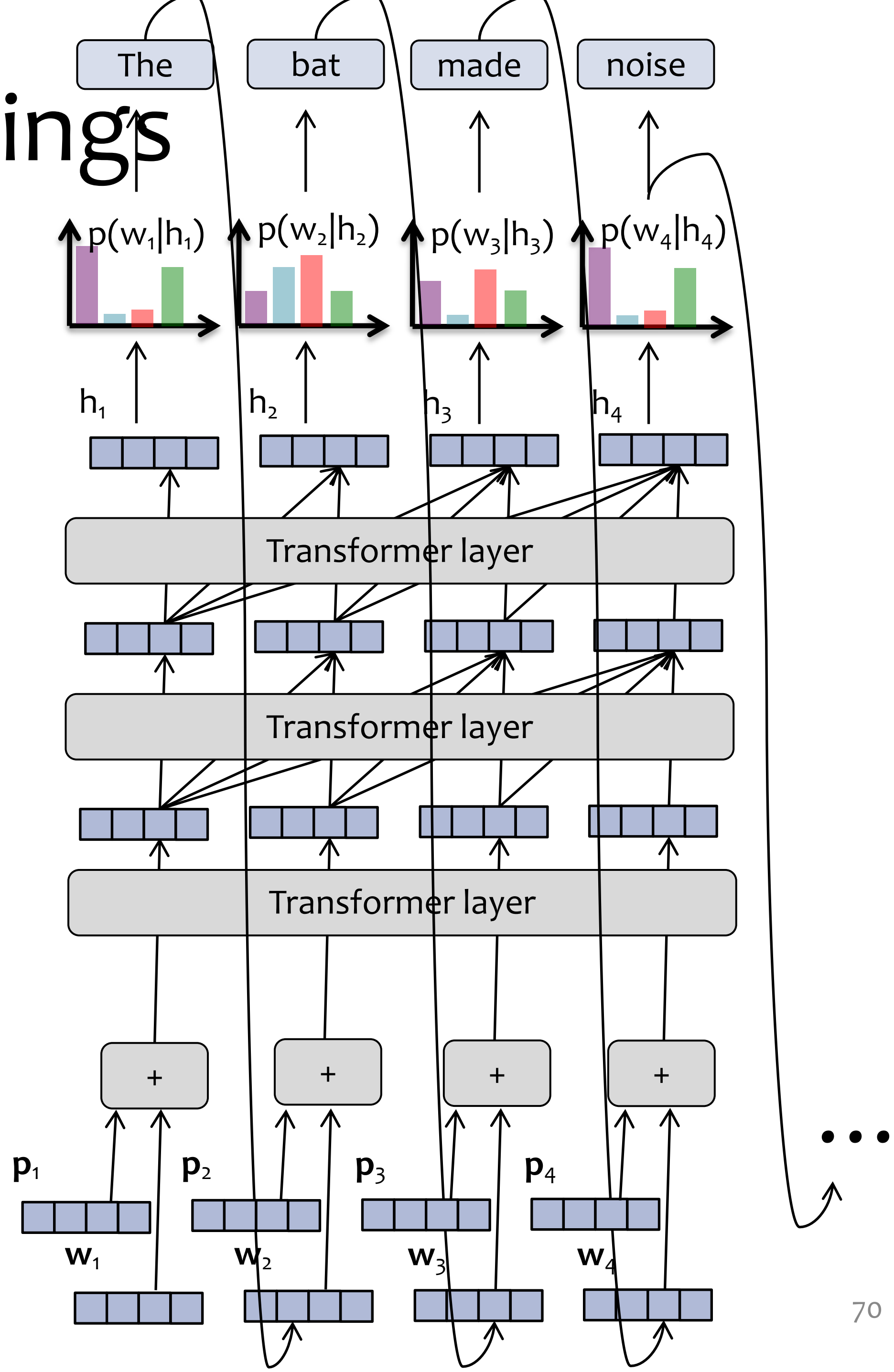
$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Answer:

Position Embeddings

- The Problem: Because attention is position invariant, we **need** a way to learn about positions
- The Solution: Use (or learn) a collection of position specific embeddings: $\mathbf{p}_t$ represents what it means to be in position t. And add this to the word embedding $\mathbf{w}_t$.
  The **key idea** is that every word that appears in position t uses the same position embedding $\mathbf{p}_t$
- There are a number of varieties of position embeddings:
  - Some are fixed (based on sine and cosine), whereas others are learned (like word embeddings)
  - Some are absolute (as described above) but we can also use relative position embeddings (i.e. relative to the position of the query vector)

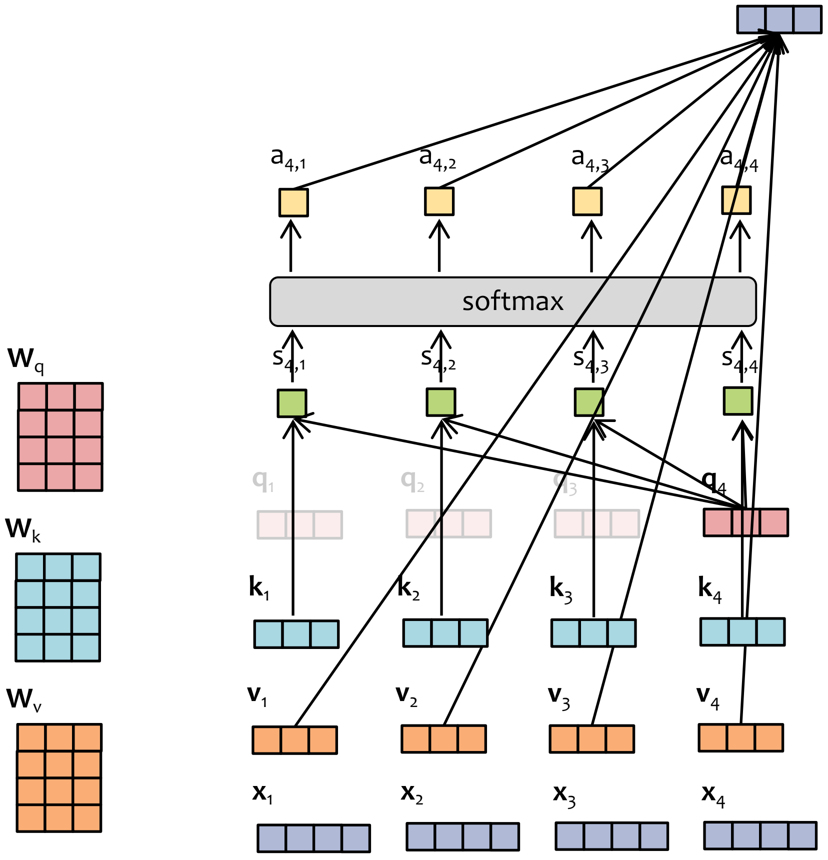


GPT-3

- GPT stands for Generative Pre-trained Transformer
- GPT is just a Transformer LM, but with a huge number of parameters

Model	# layers	dimension of states	dimension of inner states	# attention heads	# params
GPT (2018)	12	768	3072	12	117M
GPT-2 (2019)	48	1600	--	--	1542M
GPT-3 (2020)	96	12288	4*12288	96	175000M

Matrix Version of Scaled Dot-Product Attention



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

$$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$$ attention weights

$$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$$ scores

$$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$$ queries

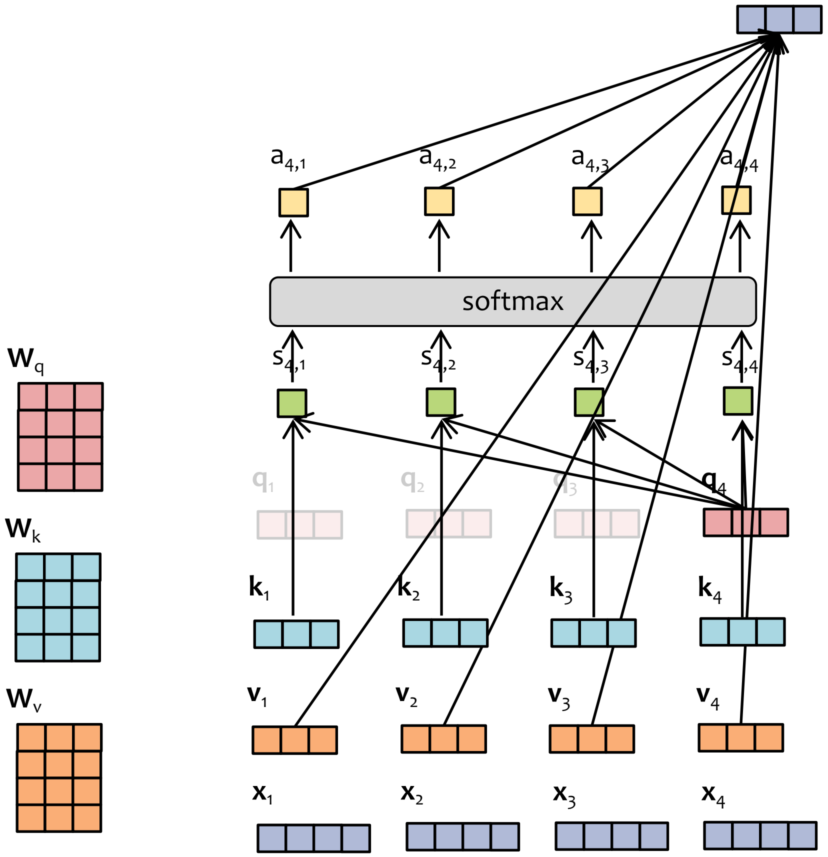
$$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$$ keys

$$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$$ values

- For speed, we compute all the queries at once using matrix operations
- First we pack the queries, keys, values into matrices:
  - $Q = [q_1,\ldots,q_N]^T$
  - $K = [k_1,\ldots,k_N]^T$
  - $V = [v_1,\ldots,v_N]^T$
- Then we compute all the queries at once:

$$\text{Attn}(\mathbf{x}_{1:N}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}V\right)$$

Matrix Version of Scaled Dot-Product Attention



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$ attention weights

$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

- For speed, we compute all the queries at once using matrix operations
- First we pack the queries, keys, values into matrices:
  - $Q = [q_1, \ldots, q_N]^T$
  - $K = [k_1, \ldots, k_N]^T$
  - $V = [v_1, \ldots, v_N]^T$
- Then we compute all the queries at once:

$$\text{Attn}(\mathbf{x}_{1:N}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}V\right)$$

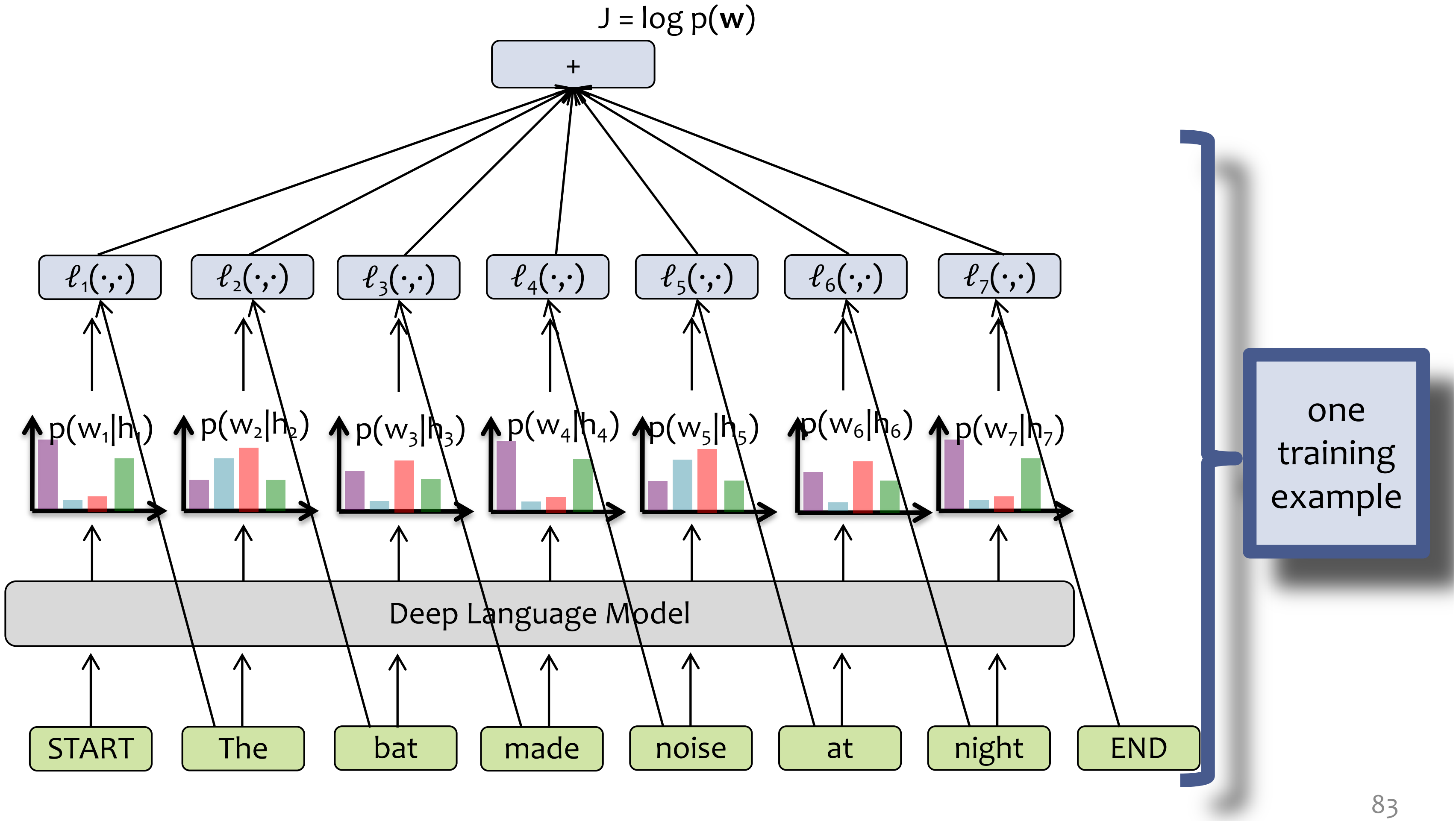
In practice, the attention weights are computed for all time steps T, then we mask out (by setting to –inf) all the inputs to the softmax that are for the timesteps to the right of the query.

LEARNING A TRANSFORMER LM

Learning a Deep Language Model

- Each training example is a sequence (e.g. sentence), so we have training data D = {$\mathbf{w}^{(1)}$, $\mathbf{w}^{(2)}$, … , $\mathbf{w}^{(N)}$}
- The objective function for a Deep LM (e.g. RNN-LM or Tranformer-LM) is typically the log-likelihood of the training examples:
  $$J(\boldsymbol{\theta}) = \Sigma_i \log p_{\boldsymbol{\theta}}(\mathbf{w}^{(i)})$$
- We train by mini-batch SGD (or your favorite flavor of mini-batch SGD)

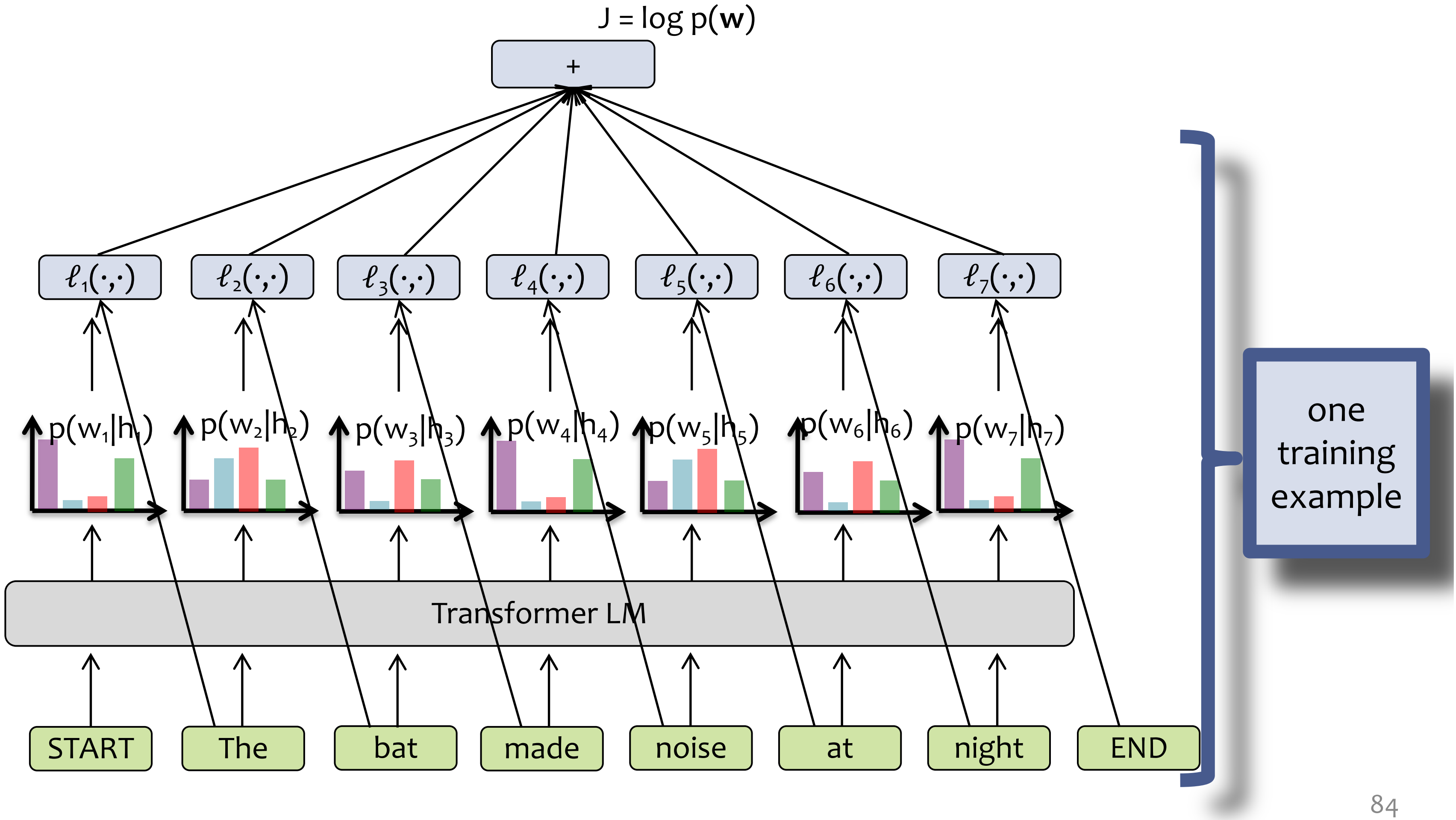
$$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \ldots , w_T)$$
$$= \log p(w_1 \mid h_1) + \log p(w_2 \mid h_2) + \ldots + \log p(w_2 \mid h_T)$$



Learning a Transformer Language Model

- Each training example is a sequence (e.g. sentence), so we have training data D = $\{\mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \ldots, \mathbf{w}^{(N)}\}$
- The objective function for a Deep LM (e.g. RNN-LM or Tranformer-LM) is typically the log-likelihood of the training examples:
  $J(\boldsymbol{\theta}) = \Sigma_i \log p_{\boldsymbol{\theta}}(\mathbf{w}^{(i)})$
- We train by mini-batch SGD (or your favorite flavor of mini-batch SGD)

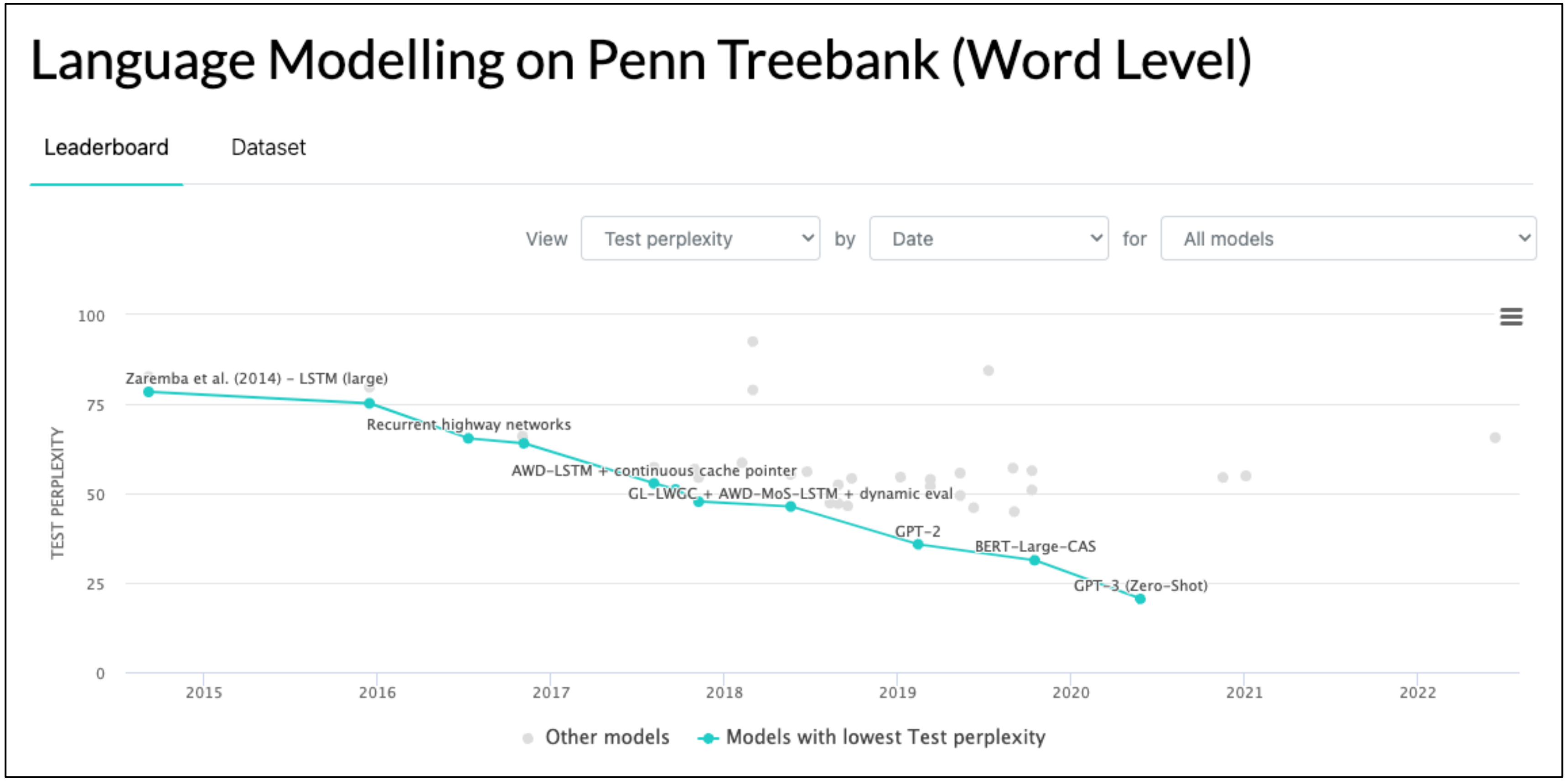
$$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \ldots , w_T)$$
$$= \log p(w_1 \mid h_1) + \log p(w_2 \mid h_2) + \ldots + \log p(w_2 \mid h_T)$$



Language Modeling

An aside:

- State-of-the-art language models currently tend to rely on **transformer networks** (e.g. GPT-2)
- RNN-LMs comprised most of the early neural LMs that **led to** current SOTA architectures



Language Modelling on Penn Treebank (Word Level)

Leaderboard Dataset

View Test perplexity by Date for All models

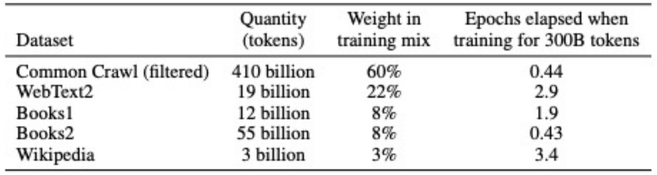
TEST PERPLEXITY
100
75
50
25
0
2015
2016
2017
2018
2019
2020
2021
2022
Zaremba et al. (2014) - LSTM (large)
Recurrent highway networks
AWD-LSTM + continuous cache pointer
GL-LWGC + AWD-MoS-LSTM + dynamic eval
GPT-2
BERT-Large-CAS
GPT-3 (Zero-Shot)
● Other models
● Models with lowest Test perplexity

Figure from https://paperswithcode.com/sota/language-modelling-on-penn-treebank-word

Why does efficiency matter?

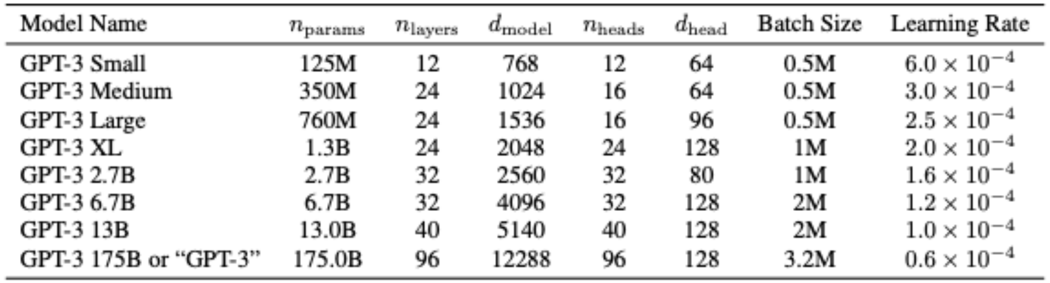
Case Study: GPT-3

- # of training tokens = 500 billion
- # of parameters = 175 billion
- # of cycles = 50 petaflop/s-days (each of which are 8.64e+19 flops)



Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Table 2.2: Datasets used to train GPT-3. "Weight in training mix" refers to the fraction of examples during training that are drawn from a given dataset, which we intentionally do not make proportional to the size of the dataset. As a result, when we train for 300 billion tokens, some datasets are seen up to 3.4 times during training while other datasets are seen less than once.



Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

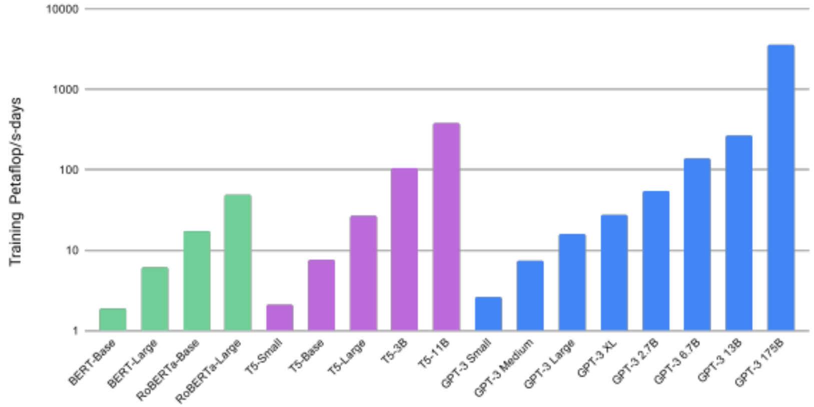


Figure 2.2: Total compute used during training. Based on the analysis in Scaling Laws For Neural Language Models [KMH+20] we train much larger models on many fewer tokens than is typical. As a consequence, although GPT-3 3B is almost 10x larger than RoBERTa-Large (355M params), both models took roughly 50 petaflop/s-days of compute during pre-training. Methodology for these calculations can be found in Appendix D.

Figure from https://arxiv.org/pdf/2005.14165.

Summary

- Task: Language Modeling
  - noisy channel models (speech / MT)
  - (historical) Large LMs (n-gram models)
- Model: GPT
  - Attention (computation graph)
  - Transformer-LM (cf. RNN-LM)
- Learning for LLMs
  - Pre-training (unsupervised learning)
  - Reinforcement Learning with Human Feedback (deep RL)
- Optimization for LLMs
  - Adam (cf. SGD)
  - Distributed training
- Societal Impacts of LLMs