Backpropagation

CSCI 601 471/671
NLP: Self-Supervised Models

https://self-supervised.cs.jhu.edu/sp2023/

[Slide credit: Andrej Karpathy and many others]
HW update

- HW1 grades are up!
  - Stats: Mean: 93.1 (std: ~5)
  - There was a mistake in grading Q4.6, but should be corrected now.

- Regrade requests can be submitted via Gradescope.
  - Please don’t spam us! 🙏

- HW3 is up!
  - Focus: training neural networks
Recap: Feel Forward Neural Networks
Recap: Feel Forward Neural Networks

Weights to learn!

Weights to learn!

Weights to learn!
Recap: Jacobian Matrix

- Generalization of gradients

- Given a function with \textit{m outputs} and \textit{n inputs}
  \[
f(x) = [f_1(x_1, x_2, \ldots, x_n), \ldots, f_m(x_1, x_2, \ldots, x_n)] \in \mathbb{R}^m
  \]

- It’s Jacobian is an \textit{m x n matrix} of partial derivatives:
  \[
  J_f(x) = \begin{bmatrix}
  \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\
  \vdots & \ddots & \vdots \\
  \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n}
  \end{bmatrix} \in \mathbb{R}^{m \times n}
  \]
Recap: Chain Rule for Multivariable Functions

- Looks similar to the single-variable setup:

\[ J_{f \circ g}(x) = J_f(g(x)) \cdot J_g(x) \]

Note, the above statement is a matrix multiplication!
Function \( f \circ g \) has \( m \) outputs and \( d \) inputs \( \rightarrow m \) by \( d \) Jacobian.
Training Neural Networks: Setup

- We are given an architecture though its weights $W$.
- We are given a loss function $\ell: \mathbb{R} \times \mathbb{R} \rightarrow (0, 1)$
  - $\ell(y^*, y)$ quantifies distance between an answer $y^*$ and prediction $y = \text{NN}(x; W)$ — lower is better
- Also given a training data $D = \{(x_i, y_i^*)\}$
- Overall objective to optimize: $\mathcal{L}(D; W) = \sum_{(x_i, y_i^*) \in D} \ell(\text{NN}(x_i; W), y_i^*)$

Weights to learn! $x = (x_1, x_2, x_3, x_4) \in \mathbb{R}^4$
Training Neural Networks ~ Optimizing Parameters

- We can use gradient descent to minimizes the loss.
- At each step, the weight vector is modified in the direction that produces the steepest descent along the error surface.

\[ x = (x_1, x_2, x_3, x_4) \in \mathbb{R}^4 \]

\[ y \in \mathbb{R} \]
Training Neural Networks ~ Optimizing Parameters

For each sub-parameter $W_i \in \mathbf{W}$:

$$W_i^{(t+1)} = W_i^{(t)} - \alpha \frac{\partial \mathcal{L}}{\partial W_i}$$

It all comes down to effectively computing $\frac{\partial \mathcal{L}}{\partial W_i}$

$x = (x_1, x_2, x_3, x_4) \in \mathbb{R}^4$

$y \in \mathbb{R}$
Training Neural Networks ~ Computing the Gradients

- How do you efficiently compute $\frac{\partial L}{\partial w_i}$ for all parameters?
- It’s easy to learn the final layer – it’s just a linear unit.
- How about the weights in the earlier layers (i.e., before the final layer)?

$x = (x_1, x_2, x_3, x_4) \in \mathbb{R}^4$

$y \in \mathbb{R}$
Necessity of a Principled Algorithm for Gradient Computation

- **Depth** gives more representational capacity to neural networks.
- However, training **deep** nets is not trivial.
- Even if we have analytical formula for each gradient, they can be tedious and **must be repeated** for each new architecture.
- The solution is “Backpropagation” algorithm!

Architecture of the BERT model with over 24 layers and millions of parameters — we will study get to this model in a few weeks!
Key Intuitions Required for BP

1. **Gradient Descent**
   - Change the weights \( W \) in the direction of gradient to minimize the error function.

2. **Chain Rule**
   - Use the chain rule to calculate the weights of the intermediate weights.

3. **Dynamic Programming (Memoization)**
   - Memoize the weight updates to make the updates faster.
A Generic Neural Network

- Given the following definition:

  \[ x = h_0 \in \mathbb{R}^{d_0} \text{ (input)} \]
  \[ h_{i+1} = f_i(W_i h_i) \in \mathbb{R}^{d_i} \text{ (hidden layer } i \text{, } 0 \leq i \leq L - 1) \]
  \[ y = u^T h_L \in \mathbb{R} \text{ (output)} \]
  \[ \mathcal{L} = \ell(y, y^*) \in \mathbb{R} \text{ (loss)} \]

- Trainable parameters: \( W = \{W_0, W_1, ..., W_L, u\} \)
Given some [initial] values for the parameters, we can compute the forward pass, layer by layer.

Forward pass is basically $L$ matrix multiplications, each followed by an activation function.

Matrix multiplication can be done efficiently with GPUs.

Therefore, forward pass is somewhat fast.

Complexity of forward pass, linear of depth $O(L)$. 
A Generic Neural Network: Direct Gradients

\[ \begin{align*}
x &= h_0 \in \mathbb{R}^{d_0} \text{ (input)} & y &= u^T h_L \in \mathbb{R} \text{ (output)} \\
h_{i+1} &= f_i(W_i h_i) \in \mathbb{R}^{d_i} & \mathcal{L} &= \ell(y, y^*) \in \mathbb{R} \text{ (loss)} \\
(0 \leq i \leq L - 1) & W = \{W_0, W_1, \ldots, W_L, u\}
\end{align*} \]

We want the gradients of \( \mathcal{L} \) with respect to model parameters.

\[ \begin{align*}
\nabla_{\mathcal{L}}(W_{L-1}) &= \left( J_{\mathcal{L}}(W_{L-1}) \right)^T = \left( J_\ell(y) J_y(h_L) J_{h_L}(W_{L-1}) \right)^T \\
\nabla_{\mathcal{L}}(W_{L-2}) &= \left( J_{\mathcal{L}}(W_{L-2}) \right)^T = \left( J_\ell(y) J_y(h_L) J_{h_L}(h_{L-1}) J_{h_{L-1}}(W_{L-2}) \right)^T \\
\vdots \\
\nabla_{\mathcal{L}}(W_0) &= \left( J_{\mathcal{L}}(W_{L-3}) \right)^T = \left( J_\ell(y) J_y(h_L) J_{h_L}(h_{L-1}) \cdots J_{h_1}(W_0) \right)^T
\end{align*} \]

In total, how many matrix multiplications are done here?

(A) \( O(L) \)  (B) \( O(L^2) \)  (C) \( O(L^3) \)  (D) \( O(\exp(L)) \)
A Generic Neural Network: Gradients with Caching/Memoization

\[ \nabla_L(W_{L-1}) = \left( J_\ell(y) J_y(h_L) J_{h_L}(W_{L-1}) \right)^T = \left( \delta_L J_{h_L}(W_{L-1}) \right)^T \]
\[ \nabla_L(W_{L-2}) = \left( J_\ell(y) J_y(h_L) J_{h_L}(h_{L-1}) J_{h_{L-1}}(W_{L-2}) \right)^T = \left( \delta_{L-1} J_{h_{L-1}}(W_{L-2}) \right)^T \]
\[ \vdots \]
\[ \nabla_L(W_0) = \left( J_\ell(y) J_y(h_L) J_{h_L}(h_{L-1}) \ldots J_{h_1}(W_0) \right)^T = \left( \delta_1 J_{h_1}(W_0) \right)^T \]

- Parameter gradients depend on the gradients of the earlier layers!
- So, when computing gradients at each layer, we don’t need to start from scratch!
- I can reuse gradients computed for higher layers for lower layers (i.e., memoization).

Let \( \delta_i \) denote Jacobian at the output of layer \( i \):
\[ \delta_i = J_\ell(y) J_y(h_L) J_{h_L}(h_{L-1}) \ldots J_{h_i}(h_{i-1}) \]
\[ \delta_i = \delta_{i+1} J_{h_i}(h_{i-1}) \]

In total, how many matrix multiplications are done here when using caching/memoization?
(A) \( O(L) \)  (B) \( O(L^2) \)  (C) \( O(L^3) \)  (D) \( O(\exp(L)) \)
A Generic Neural Network: Backward Step

- Backward step computes the gradients starting from the end to the beginning, layer by layer.

- Start by computing local gradients: $J_{h_l}(h_{l-1})$

- Use then to compute upstream gradients $\delta_L$, then $\delta_{L-1}$, then $\delta_{L-2}$, ...

- Use these to compute global gradients: $\nabla_L (W_i)$

- Computational cost as a function of depth:
  - With memoization, gradient computation is a linear function of depth $L$ (same cost as the forward process!!)
  - Without memorization, gradients computation would grow quadratic with $L$
Initialize network parameters with random values
Loop until convergence
Loop over training instances
  i. **Forward step:**
     Start from the input and compute all the layers till the end (loss $\mathcal{L}$)
  
  ii. **Backward step:**
     Compute local gradients, starting from the last layer
     Compute upstream gradients $\delta_i$ values, starting from the last layer
     Use $\delta_i$ values to compute global gradients $\nabla_{\mathcal{L}}(W_i)$ at each layer
  
  iii. **Gradient update:**
     Update each parameter: $W_i^{(t+1)} \leftarrow W_i^{(t)} - \alpha \nabla_{\mathcal{L}}(W_i)$

In practice, this step is done over batches of instances!
Computation Graph: Example

- In reality, networks are not as regular as the previous example...
Back-Prop in General Computation Graph

- What if the network does not have a regular structure? Same idea!

- Sort the nodes in topological order (what depends on what)

- Forward-Propagation:
  - Visit nodes in topological sort order and compute value of node given predecessors

- Backward-Propagation:
  - Compute local gradients
  - Visit nodes in reverse order and compute global gradients using gradients of successors
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, \ y = 5, \ z = -4 \)
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, \ y = 5, \ z = -4 \)

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
**Computation Graph: An Example**

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, \ y = 5, \ z = -4 \)
- Start with local gradients!
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, y = 5, z = -4 \)
- Start with local gradients!

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

[Slide: Stanford CS231N]
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, \ y = 5, \ z = -4 \)
- Start with local gradients!

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

[Slide: Stanford CS231N]
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, y = 5, z = -4 \)
- Start with local gradients!

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

[Slide: Stanford CS231N]
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, \ y = 5, \ z = -4 \)
- Start with local gradients!

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \ \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \ \frac{\partial f}{\partial z} = q \]

[Slide: Stanford CS231N]
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, y = 5, z = -4 \)
- Start with local gradients!

\[
q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1
\]

\[
f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q
\]
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, y = 5, z = -4 \)
- Start with local gradients!

\[
q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1
\]

\[
f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q
\]

[Slide: Stanford CS231N]
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, y = 5, z = -4 \)
- Start with local gradients!

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Chain rule:

\[ \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} \]

[Slide: Stanford CS231N]
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, \ y = 5, \ z = -4 \)
- Start with local gradients!

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Chain rule:

\[ \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} \]

[Slide: Stanford CS231N]
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, y = 5, z = -4 \)
- Start with local gradients!

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

[Slide: Stanford CS231N]
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, \ y = 5, \ z = -4 \)
- Start with local gradients!

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Chain rule:

\[ \frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} \]

[Slide: Stanford CS231N]
Computation Graph: An Example

\[ f(x, y, z) = (x + y)z \]

- Evaluated at: \( x = -2, y = 5, z = -4 \)
- Start with local gradients!

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Chain rule:

\[ \frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} \]

[Slide: Stanford CS231N]
A Generic Example
Figure from Andrej Karpathy
"local gradient"

\[ \frac{\partial z}{\partial x} \]

\[ \frac{\partial z}{\partial y} \]

\[ \frac{\partial L}{\partial z} \]

"Upstream gradient"

Figure from Andrej Karpathy
\[ \frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x} \]

“Downstream gradients”

“Local gradient”

“Upstream gradient”

Figure from Andrej Karpathy
"Local gradient" = \( \frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x} \)

"Downstream gradients" = \( \frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y} \)

"Upstream gradient" = \( \frac{\partial L}{\partial z} \)

Figure from Andrej Karpathy
Figure from Andrej Karpathy
Demo time!

- Link: https://playground.tensorflow.org/
Chapter Plan

1. Feed-forward networks
2. Neural nets: brief history
3. Word2Vec as a simple neural network
4. Training neural networks: back-propagation
5. Backprop in practice
Backprop in PyTorch

\[ f(x, y, z) = (x + y)z \]

Want: \[
\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}
\]

```python
x = torch.tensor(-2.0, requires_grad=True)
y = torch.tensor(5.0, requires_grad=True)
z = torch.tensor(-4.0, requires_grad=True)

f = (x+y)*z  # Define the computation graph

f.backward()  # PyTorch's internal backward gradient computation

print('Gradients after backpropagation:', x.grad, y.grad, z.grad)
```
PyTorch’s Implementation: Forward/Backward API

- PyTorch has implementation of forward/backward operations for various operators.
- Example: multiplication operator

```python
class Multiply(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, y):
        ctx.save_for_backward(x, y)
        z = x * y
        return z
    @staticmethod
    def backward(ctx, grad_z):
        x, y = ctx.saved_tensors
        grad_x = y * grad_z  # dz/dx * dL/dz
        grad_y = x * grad_z  # dz/dy * dL/dz
        return grad_x, grad_y
```
PyTorch Operators

PyTorch’s lower-level functions translate activities to graphics processor via libraries like OpenGL.
Example Activation Functions

```cpp
#define PRECISION $precision
#define FORMAT $format

layout(std430) buffer;

#pragma Qualifiers: layout - storage - precision - memory */

layout(set = 0, binding = 0, FORMAT) uniform PRECISION restrict writeonly image3D u0;
layout(set = 0, binding = 1) uniform PRECISION sampler3D u1;
layout(set = 0, binding = 2) uniform PRECISION restrict                     Block {
  ivec4 size;
  } uBlock;

layout(local_size_x = 0, local_size_y = 1, local_size_z = 2) in;

void main() {
  const ivec3 pos = ivec3(gl_GlobalInvocationID);
  if (all(lessThan(pos, uBlock.size.xyz))) {
    imageStore(uOutput, pos, 1/(1+exp(-1*texture(uInput, pos, 0))));
  }
}
```
Why Learn All These Details About Backprop?

● Modern deep learning frameworks compute gradients for you!

● But why take a class on compilers or systems when they are implemented for you?
  ○ Understanding what is going on under the hood is useful!

● Backpropagation doesn’t always work perfectly out of the box
  ○ Understanding why is crucial for debugging and improving models
Backprop in Practice
Activation Functions

● How do you choose what activation function to use?
● In general, it is problem-specific and might require trial-and-error.
● Here are some tips about popular action functions.
Activation Functions: Sigmoid

- Squashes numbers to range \([0, 1]\)
- Historically popular, interpretation as “firing rate” of a neuron
- Key limitation: Saturated neurons “kill” the gradients
- Whenever \(|x| > 5\), the gradients are basically zero.

\[
s(x) = \frac{1}{1 + e^{-x}}
\]

\[
\frac{\partial s(x)}{\partial x} = s(x) (1 - s(x))
\]

If all the gradients flowing back will be zero and weights will never change.

[dance figure: https://www.imaginary.org/gallery/maths-dance-moves]
Activation Functions: Tanh

- Symmetric around [-1, 1]
- Still saturates $|x| > 3$ and “kill” the gradients
- Zero-centered — good for stacking hidden layers

[dance figure: https://www.imaginary.org/gallery/maths-dance-moves]

[LeCun et al., 1991]
Activation Functions : ReLU

- Computationally efficient
- In practice, converges faster than sigmoid/tanh in practice
- Does not saturate (in +region) — will die less!

ReLU (Rectified Linear Unit)

[Krizhevsky et al., 2012]
Activation Functions: Leaky ReLU

- Does not saturate — will not die.
- Computationally efficient
- In practice it converges faster than sigmoid/tanh in practice

- Other parametrized variants:
  - Parametric Rectifier (PReLU): \( f(x) = \max(\alpha x, x) \) [He et al., 2015]
  - Maxout: \( \max(w_1^T x + b_1, w_2^T x + b_2) \) [Goodfellow et al., 2013]

- Provide more flexibility, though at the cost of more learnable parameters.
  - For example, Maxout doubles the number of parameters.

\[ f(x) = \max(0.01x, x) \]
How do You Choose What Activation Function to Use?

- In general, it is problem-specific and might require trial-and-error.

- A useful recipe:
  1. Generally, ReLU is a good activation to start with.
  2. Time/compute permitting, you can try other activations to squeeze out more performance.
Exploding/Vanishing Gradients

- Remember gradient computation at layer $L - k$:

$$\nabla_L (W_{L-k}) = \left( J_y (h_L) J_{h_{L-1}} (h_{L-2}) ... J_{h_{L-k+1}} (W_{L-k}) \right)^T$$

- This matrix multiplication could quickly approach
- $\infty$, if the matrix elements are large — exploding gradients.
- $0$, if the matrix elements are small — vanishing gradients.

- For those interested, convergences of matrix powers is determined by its largest eigenvalue (out of scope for this class, extra credit).

- $\infty/0$ gradients would kill learning (no flow of information).
Residual Connections/Blocks

- Create direct “information highways” between layers.

- Shown to diminish the effect of vanishing/exploding gradients
  - Early in the training, there are fewer layers to propagate through.
  - The network would restore the skipped layers, as it learns richer features.
  - It is also shown to make the optimization objective smoother.
  - Fun fact: the paper introducing residual layers (He et al. 2015) is the most cited paper of century.

[Li et al. “Visualizing the Loss Landscape of Neural Nets”]
Weight Initialization

- Initializing all weights with a fixed constant (e.g., 0) is a very bad idea! (why?)

- If the neurons start with the same weights, then all the neurons will follow the same gradient, and will always end up doing the same thing as one another.
Weight Initialization

- Better idea: initialize weights with random Gaussian noise.

```python
x = torch.tensor.empty(3, 5)
nn.init.normal_(w)
```

- There are fancier initializations (Xavier, Kaiming, etc.) that we won’t get into.

[read more here: https://pytorch.org/docs/stable/nn.init.html]
Comments on Training

- No guarantee of convergence; neural networks form non-convex functions with multiple local minima.
- In practice, many large networks can be trained on large amounts of data for realistic problems.
- May be hard to set learning rate and to select number of hidden units and layers.
- Many steps (tens of thousands) may be needed for adequate training. Large data sets may require many hours of CPU.
- Termination criteria: Number of epochs; Increased error on a validation set.
- To avoid local minima: several trials with different random initial weights with majority or voting techniques.
Over-training Prevention

- Running too many epochs and/or a NN with many hidden layers may lead to an **overfit** network.
- Keep a **held-out validation** set and evaluate accuracy after every epoch.
- Early stopping: maintain weights for best performing network on the validation set and return it when performance decreases significantly beyond that.
- To avoid losing training data to validation:
  - Use 10-fold cross-validation to determine the average number of epochs that optimizes validation performance.
  - Train on the full data set using this many epochs to produce the final results.
Over-fitting prevention

- Too few hidden units prevent the system from adequately fitting the data and learning the concept.
- Using too many hidden units leads to over-fitting.
- Similar cross-validation method can be used to determine an appropriate number of hidden units. (general)
- Another approach to prevent over-fitting is weight-decay: all weights are multiplied by some fraction in (0,1) after every epoch.
  - Encourages smaller weights and less complex hypothesis
  - Equivalently: change Error function to include a term for the sum of the squares of the weights in the network. (general)
Dropout training

- In each forward pass, randomly set some neurons to zero
- Probability of dropping is a hyperparameter; 0.5 is common
- Dropout is implicitly a ensemble (average) of model that share parameters.
  - Each binary mask is one model
  - For example, an FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
  - Only $\sim 10^{82}$ atoms in the universe ...

Dropout During Test Time

- The issue for test time is that Dropout adds randomization.
  - Each dropout mask would lead to a slightly different outcome.
- In ideal world, we would like to “average out” the outcome across all the possible random masks:
  - Not feasible.
- The alternative is to not apply dropout. Without dropout, the input values to each neuron would be higher than what was seen during the training (mismatch between train/test).
  - Example: Input to activation during:
    - training time: \( E[a] = \frac{1}{4}(w_1 x_1 + w_2 x_2) + \frac{1}{4}(0 + 0) \)
      \[ + \frac{1}{4}(0 + w_2 x_2) + \frac{1}{4}(w_1 x_1 + 0) = \frac{1}{2}(w_1 x_1 + w_2 x_2) \]
    - test time: \( E[a] = w_1 x_1 + w_2 x_2 \)
- Solution: scale the values proportional to dropout probability.
  - Can be applied in either testing (scaling down) or training (scaling up).
Dropout in Practice

Just call the PyTorch function!

It automatically
- activates the dropout for training.
- deactivates it during evaluations and scales the values according to its parameter.

```
dropout = nn.Dropout(p=0.2)
x = torch.randn(20, 16)
y = dropout(x)
```

```
# training step
...
model.train()
...

# evaluate model:
...
model.eval()
...
```

The Only Time You Want to Overfit: The First Tryout

- A model with buggy implementation (e.g., incorrect gradient calculations or updates) cannot learn anything.
- Therefore, a good and easy sanity check is to see if you can overfit few examples.
  - This is really the first test you should do, before any hyperparameter tuning.
- Try to train to 100% training accuracy/performance on a small sample (<30) of training data and monitor the training loss trends.
  - Does it down? If not, something must be wrong.
  - Try checking the learning rate or modifying the initialization.
  - If those don’t help, check the gradients.
    - If they’re NaN or Inf, might indicate exploding gradients.
    - If they’re zeros, might indicate vanishing gradients.
Demo Time!

- [https://teachablemachine.withgoogle.com/](https://teachablemachine.withgoogle.com/)
Chapter Summary

- Feed-forward network architecture
- Word2Vec is just a feedforward net!
  - And we can easily extend it!
- We learned Back-Prop, the most important algorithm in neural networks!
  - Recursively (and hence efficiently) apply the chain rule along computation graph
- Lots of empirical tricks for training neural networks:
  - First test: check if you can overfit.
  - Dropout
  - Be mindful of activations
  - Careful of exploding/vanishing gradients