Training Neural Nets

CSCI 601 471/671
NLP: Self-Supervised Models

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

[Slide credit: Andrej Karpathy and many others]

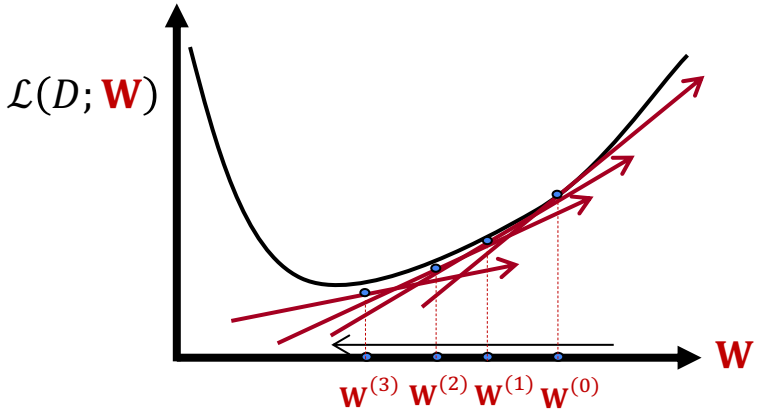
Logistics

- Extra credit:
  - HW grades are out of 100%.
  - Gradescope might show >100 because of extra credits.

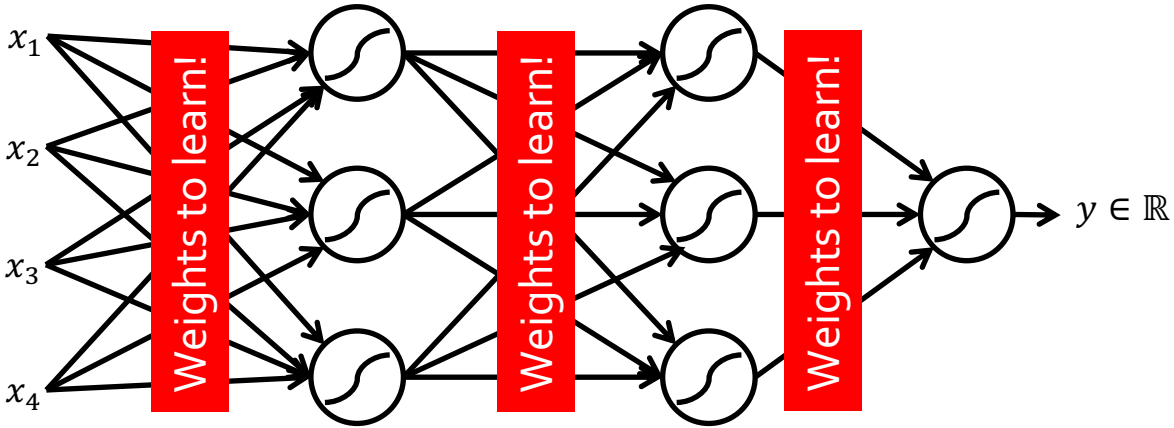
- Midterm date: Tuesday March 7
  - In class, on paper
  - Purpose: your understanding of ideas presented in the first half of the semester
  - Based on: the lectures and weekly homework assignments
  - Scope: until the end of “Transformers”

Recap: 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.



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



Recap: Back Propagation for Generic Neural Network:

Initialize network parameters with random values

Loop until convergence

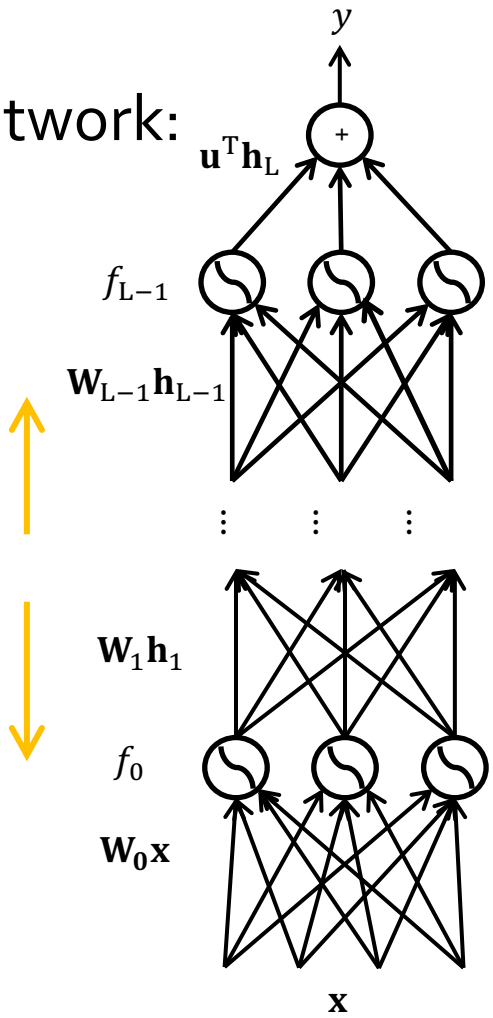
Loop over training instances

In practice, this step is done over **batches** of 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 δ_i values, starting from the last layer
Use δ_i values to compute global gradients $\nabla_{\mathcal{L}}(\mathbf{W}_i)$ at each layer

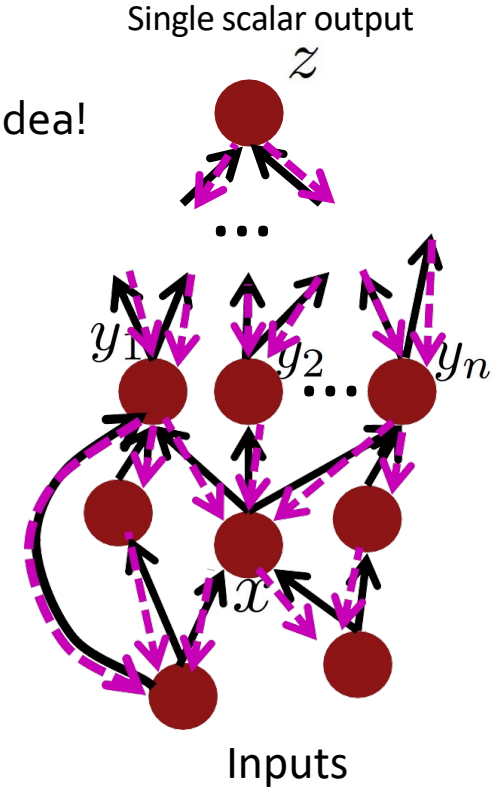
iii. **Gradient update:**
Update each parameter: $\mathbf{W}_i^{(t+1)} \leftarrow \mathbf{W}_i^{(t)} - \alpha \nabla_{\mathcal{L}}(\mathbf{W}_i)$



Recap: Backprop in General Computation Graph

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

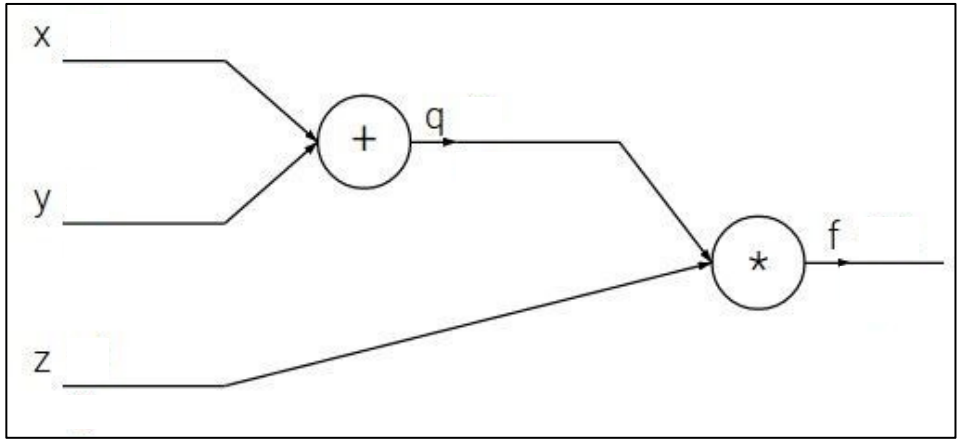
- Sort the nodes in topological order (what depends on what)
- Forward-Propagation:
  - Visit nodes in topological sort order and compute value of node given predecessors
- Backward-Propagation:
  - Compute local gradients
  - Visit nodes in reverse order and compute global gradients using gradients of successors



Recap: Backprop in PyTorch

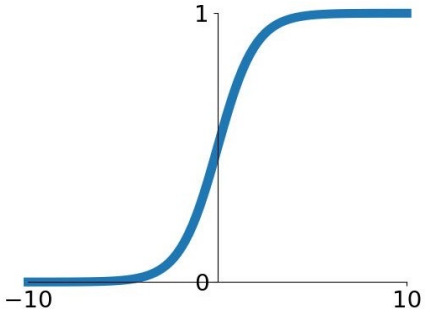
$$f(x, y, z) = (x + y)z$$

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

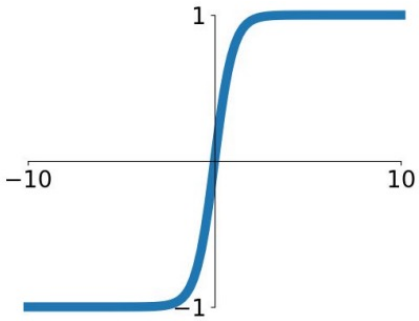



```
x = torch.tensor(-2.0, requires_grad=True)
y = torch.tensor(5.0, requires_grad=True)
z = torch.tensor(-4.0, requires_grad=True)

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)
```

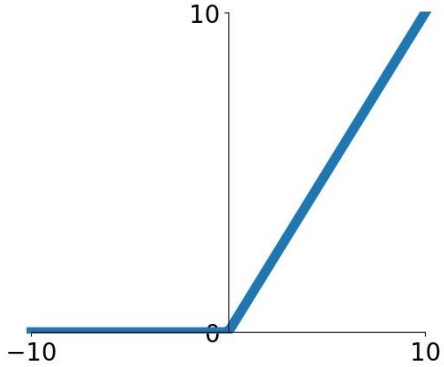
Recap: Activation Function Pros/Cons



Sigmoid



Tanh



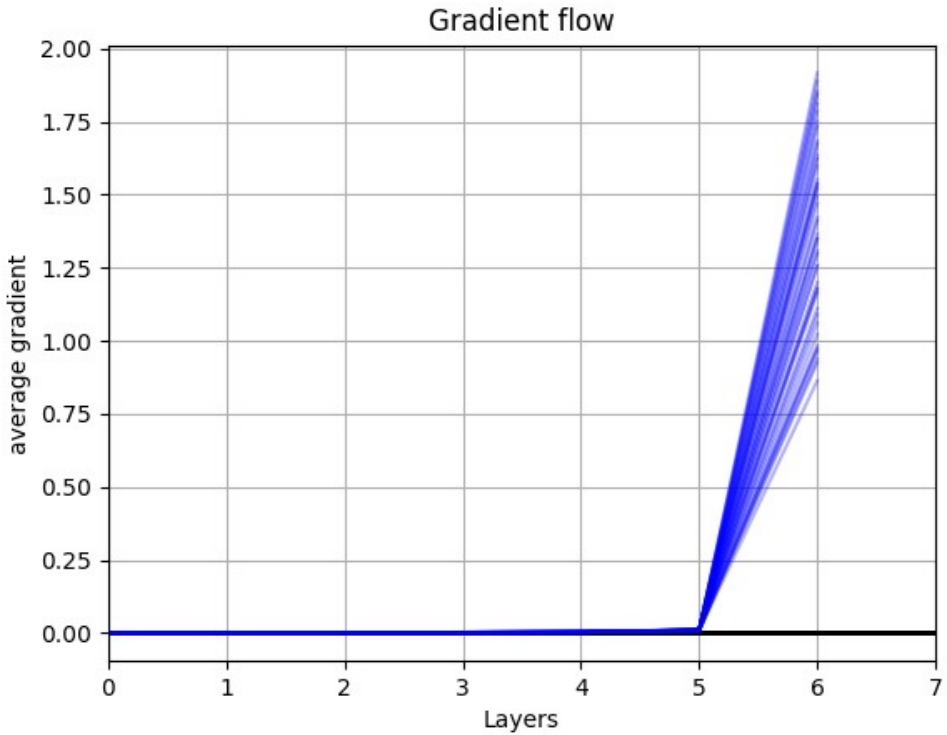
ReLU

- 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

- If many numbers |x| > 1 get multiplied, the result will become too big.
- NaN gradients --> no learning!

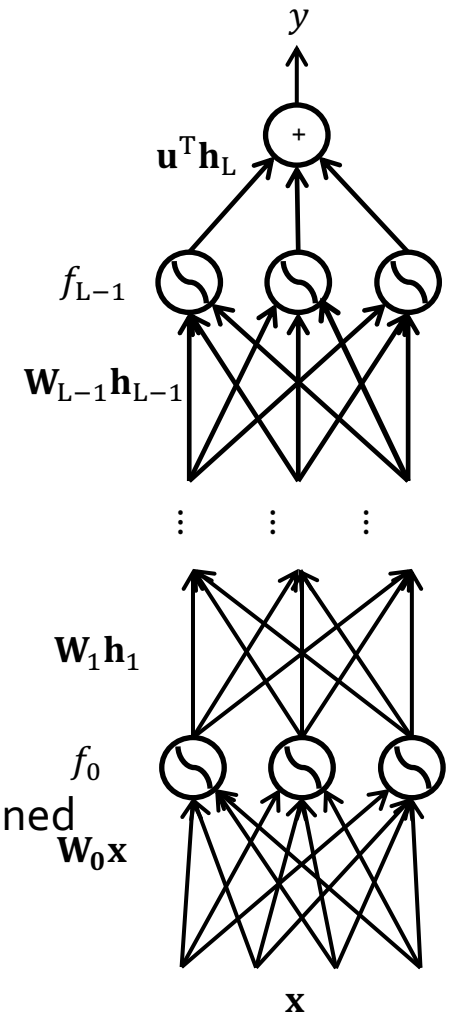
- If many numbers |x| < 1 get multiplied, the result will become too small.
- Zero gradients -> no learning!



Exploding/Vanishing Gradients

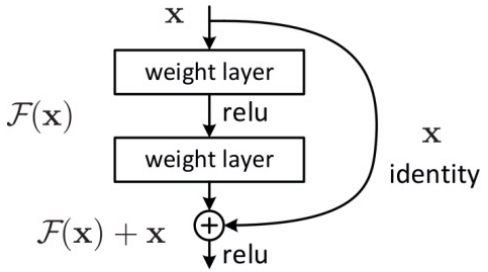
- Remember gradient computation at layer $L-k$:

$$\nabla_{\mathcal{L}}(\mathbf{W}_{L-k}) = \underbrace{\left(\mathbf{J}_{\ell}(\mathrm{y})\,\mathbf{J}_{y}(\mathbf{h}_{\mathrm{L}})\,\mathbf{J}_{\mathbf{h}_{\mathrm{L}}}(\mathbf{h}_{\mathrm{L}-1})\,\mathbf{J}_{\mathbf{h}_{\mathrm{L}-1}}(\mathbf{W}_{\mathrm{L}-2})\,\dots\,\mathbf{J}_{\mathbf{h}_{L-k+1}}(\mathbf{W}_{L-k})\right)^{\mathrm{T}}}_{\text{O(k)-many matrix multiplication}}$$



- This matrix multiplication could quickly approach
  - ∞, if the matrix elements are a large — exploding gradients.
  - 0, if the matrix elements are small — vanishing gradients.
  - $\infty/0$ gradients would kill learning (no flow of information).
- For those interested, convergences of matrix powers is determined by its largest eigenvalue (HW, extra credit).

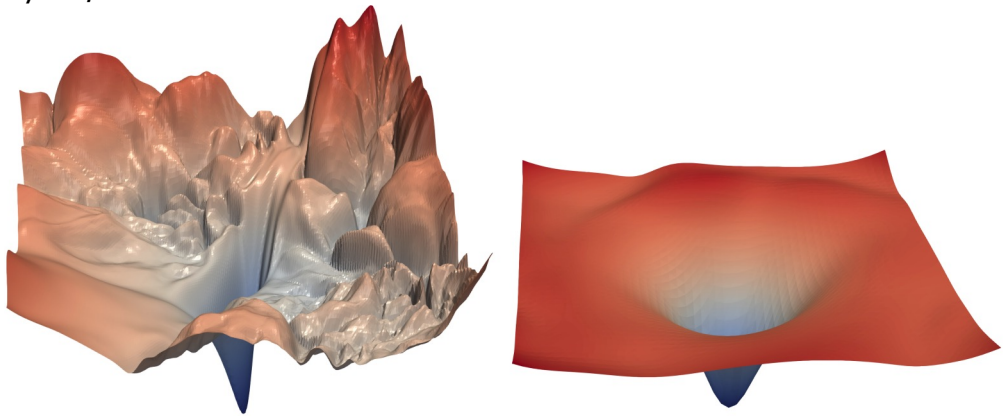
Residual Connections/Blocks



- Create direct "information highways" between layers.

- Shown to diminish 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 (He et al. 2015) introducing residual layers is the most cited paper of century!!]



(a) without skip connections (b) with skip connections

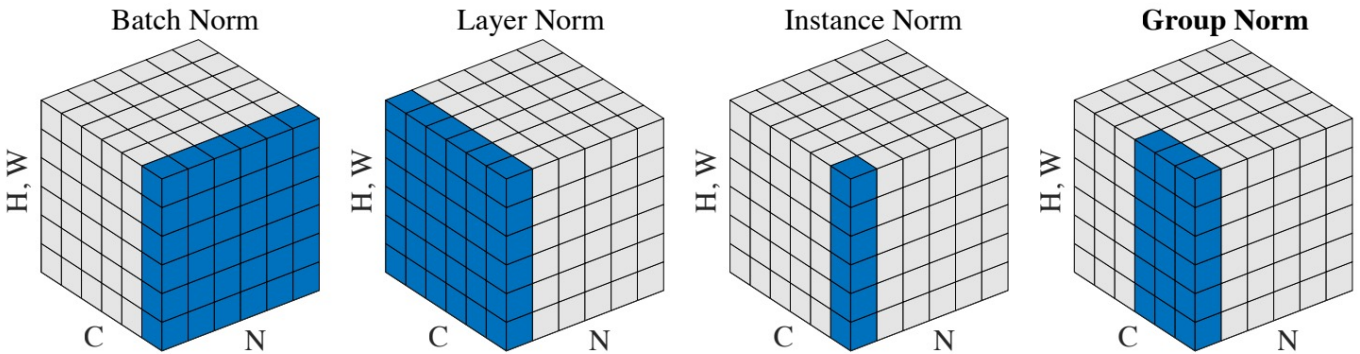
[Li et al. "Visualizing the Loss Landscape of Neural Nets"]

Normalization: Layer, Batch, …

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

- Normalization of values standardizes the ranges of values
- Prevents value disparities
- Stabilizes and speeds up training

See PyTorch documentations: https://pytorch.org/docs/stable/nn.html#normalization-layers

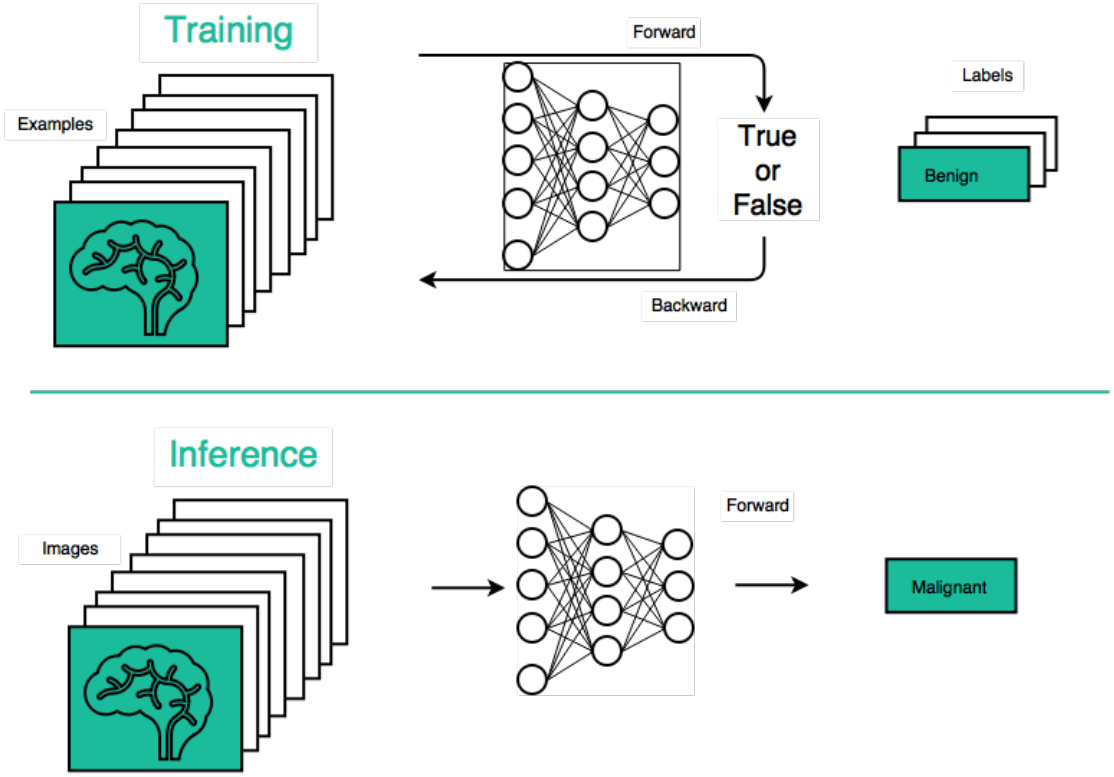


[Ba et al. "Layer Normalization"]

Batching

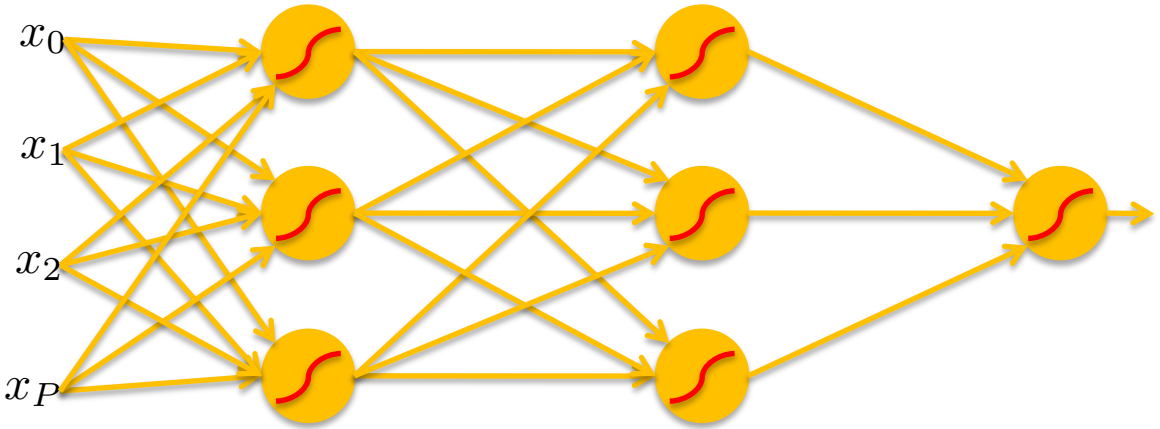
- GPUs are fast with Tensor operations

- Rather than visiting instances in sequentially , batch them together for faster training and inference.



Weight Initialization

- Initializing all weights with a fixed constant (e.g., o's) 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.
- Effective initialization is one that breaks such "symmetries" in the weight space.

Weight Initialization

- Better idea: initialize weights with random Gaussian noise.

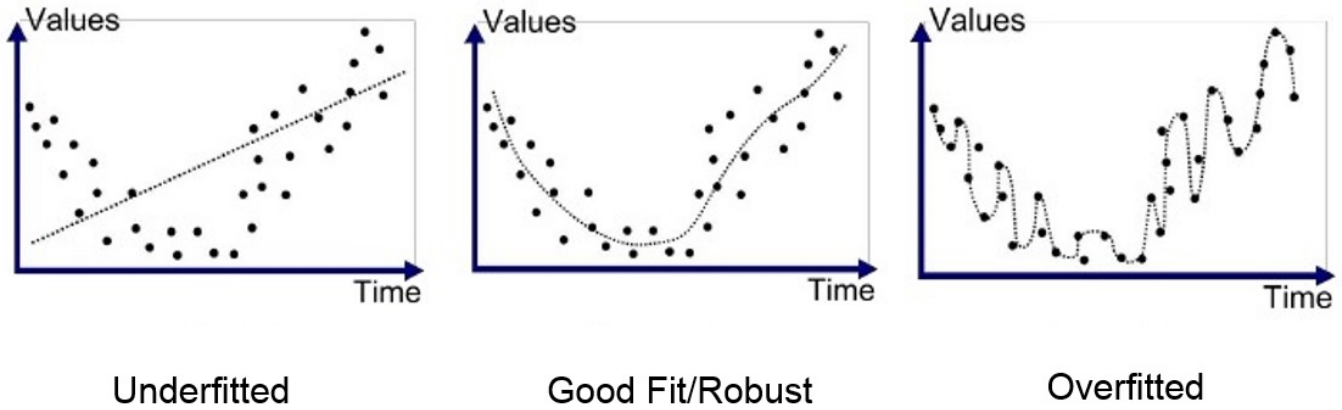
```
x = torch.tensor.empty(3, 5)
nn.init.normal_(w)
```

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

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

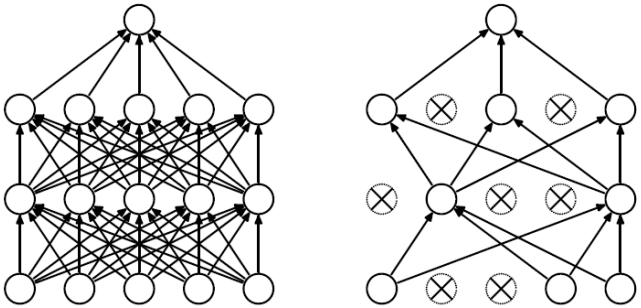
Over-training Prevention

- 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.

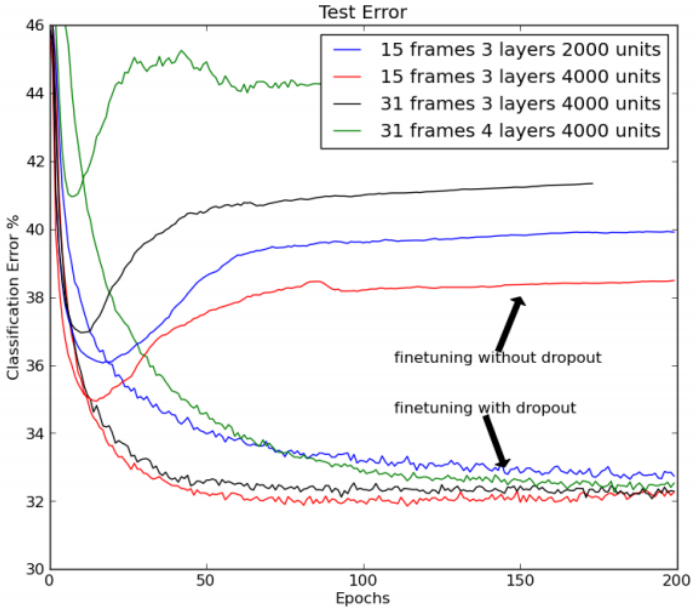


Dropout Training

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

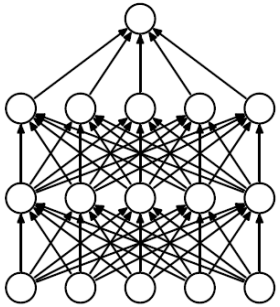


(a) Standard Neural Net (b) After applying dropout.

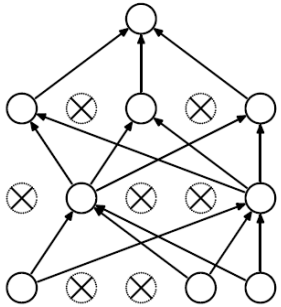


[Hinton et al, 2012; Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014]

Dropout During Test Time



(a) Standard Neural Net

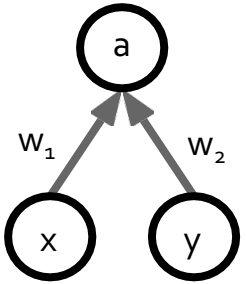


(b) After applying dropout.

- The issue for the test time:
  - 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.
  - Remember the example: a layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
  - Only ~ 10^{82} atoms in the universe …

Dropout During Test Time (2)

- 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:** imagine we apply dropout (p=0.5) to the following model:
  - Training time: $E[a] = \frac{1}{4}(w_1x_1 + w_2x_2) + \frac{1}{4}(0 + 0) + \frac{1}{4}(0 + w_2x_2) + \frac{1}{4}(w_1x_1 + 0) = \frac{1}{2}(w_1x_1 + w_2x_2)$
  - Test time: $E[a] = w_1x_1 + w_2x_2$



- **Solution:** scale the values proportional to dropout probability.
  - Can be applied in either testing (scaling down) or training (scaling up).
  - A very common interview question! ☺

Dropout in Practice

Just call the PyTorch function!

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

It automatically

- activates the dropout for training.

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

- deactivatives it during evaluations and scales the values according to its parameter.

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

See PyTorch documentations https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html

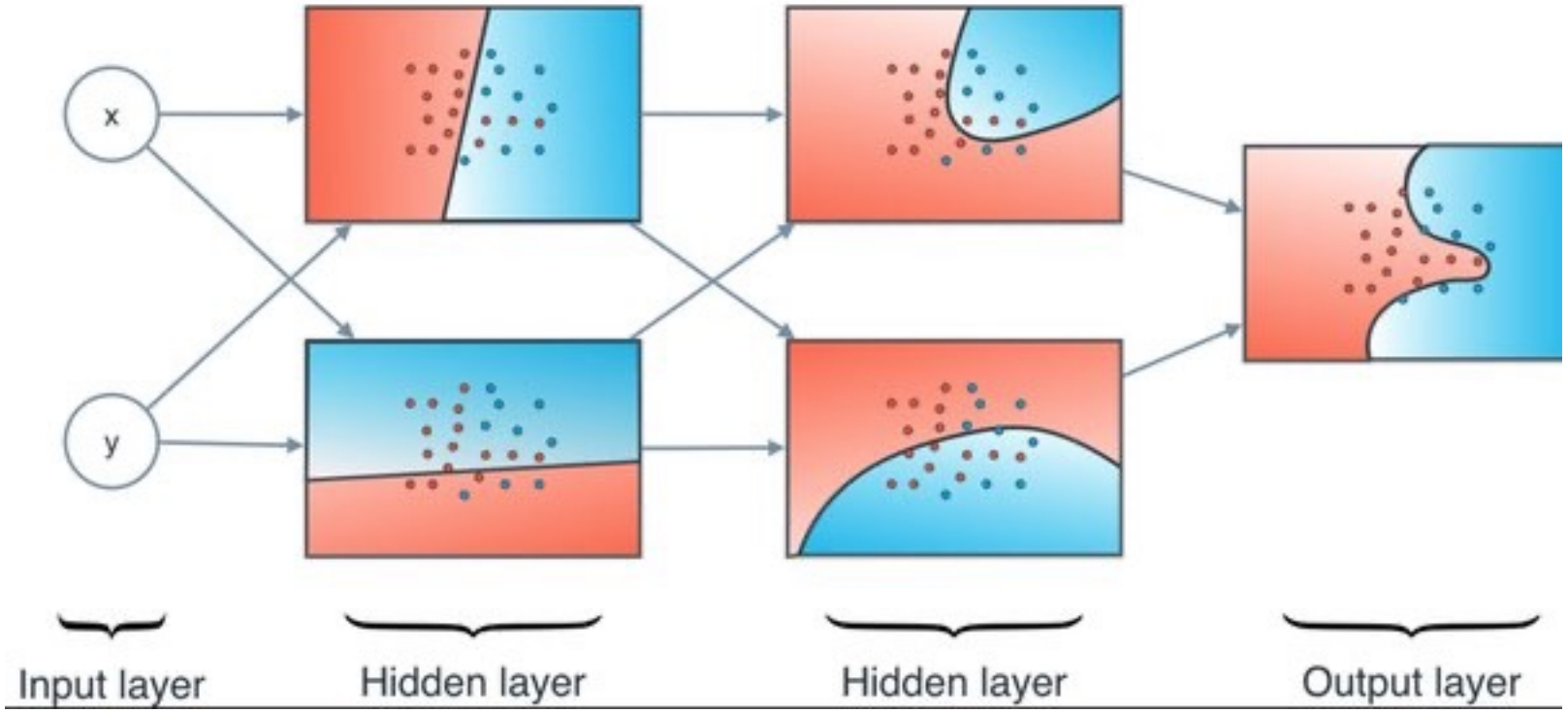
The Only Time You Want to Overfit: The First Tryout

- 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.

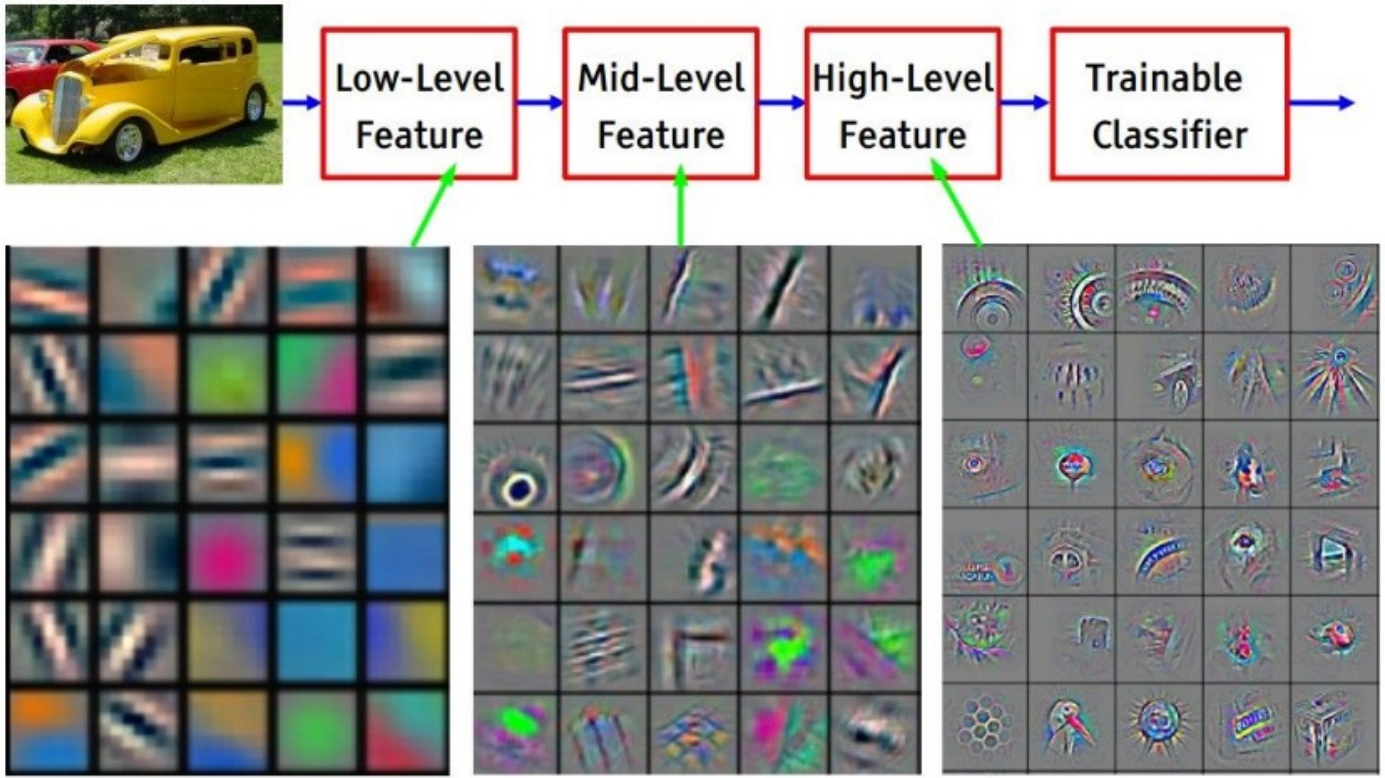
Additional Comments on Training

- No guarantee of convergence; neural networks form non-convex functions with multiple local minima
- In practice, many large networks can be trained on large data.
- Many steps (tens of thousands) may be needed for adequate training.
- May be tricky to set learning rate or number of hidden units/layers.
- To avoid local minima: several trials with different random initial weights with majority or voting techniques

Intuition about Neural Net Representations

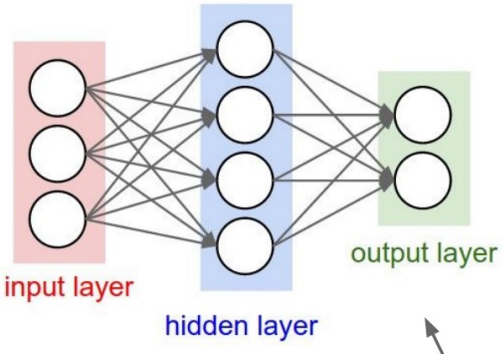


Intuition about Neural Net Representations



[Zeiler & Fergus 2013; Yosinski et al. 2015]

Neural Networks: Summary



- Feed-forward network architecture
- Word2Vec is just a feedforward net!
  - And we can easily extend it!

- We learned Backprop, 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