Feedforward Neural Nets and PyTorch Lightning

CS 759/859 Natural Language Processing Lecture 9

Samuel Carton, University of New Hampshire

Last lecture

PyTorch: Machine learning Legos

Mini-batch gradient descent

- Batch size very important

Training loop

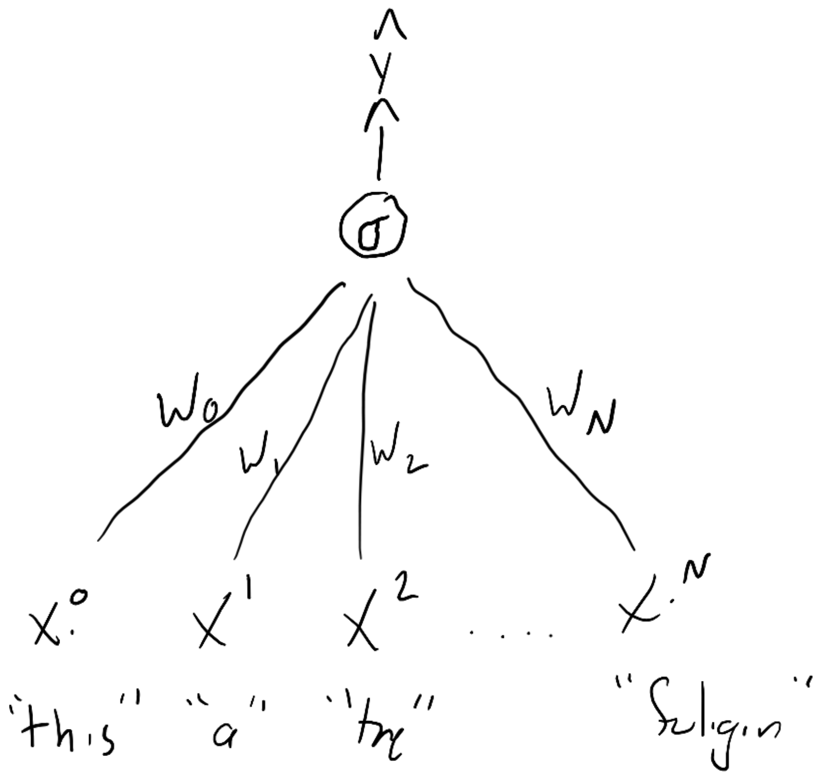
- Key elements: optimizer.zero_grad(), train_loss.backward(), optimizer.step()

Avoid overfitting by:

- Regularization
- Early stopping

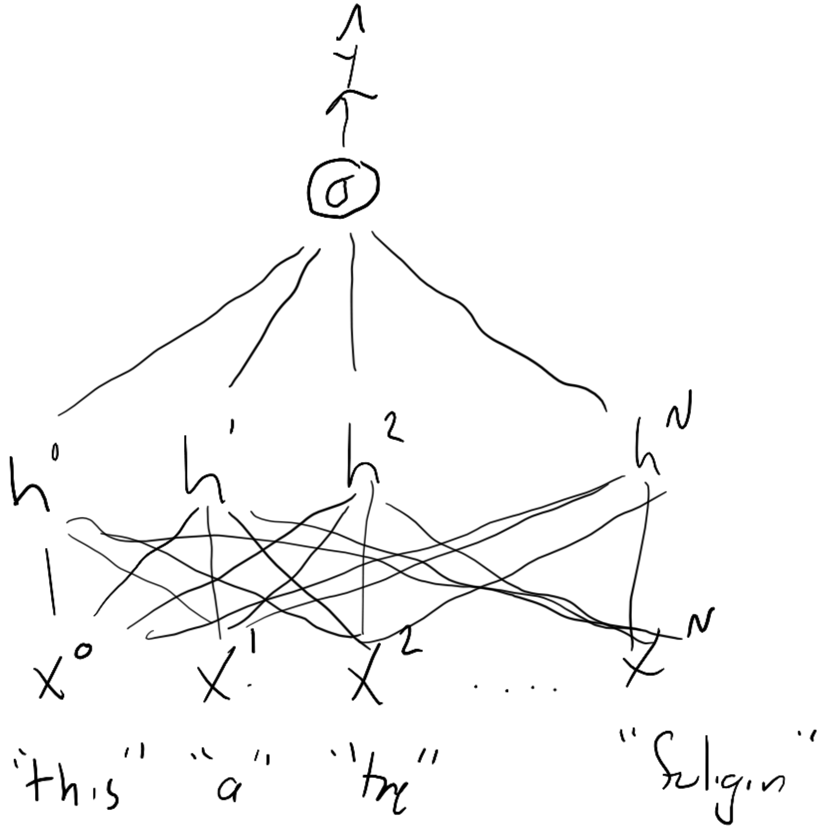
Feedforward neural nets

We’ve been working with models that look like this:



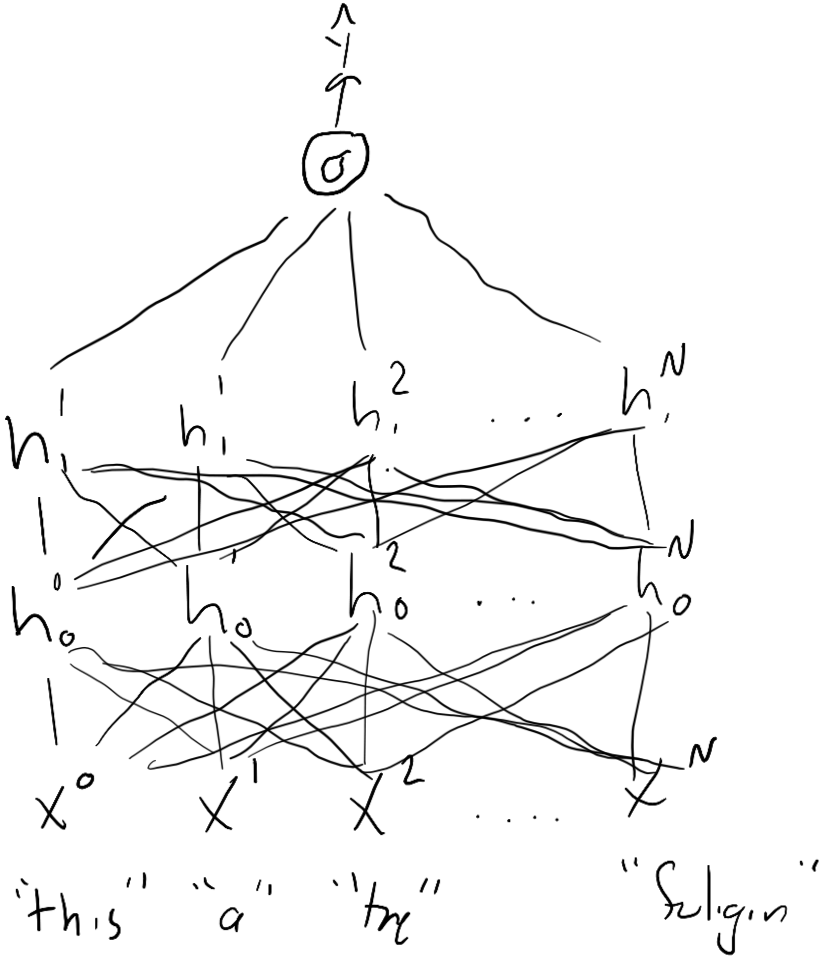
Feedforward neural nets

But what about models that look like this:



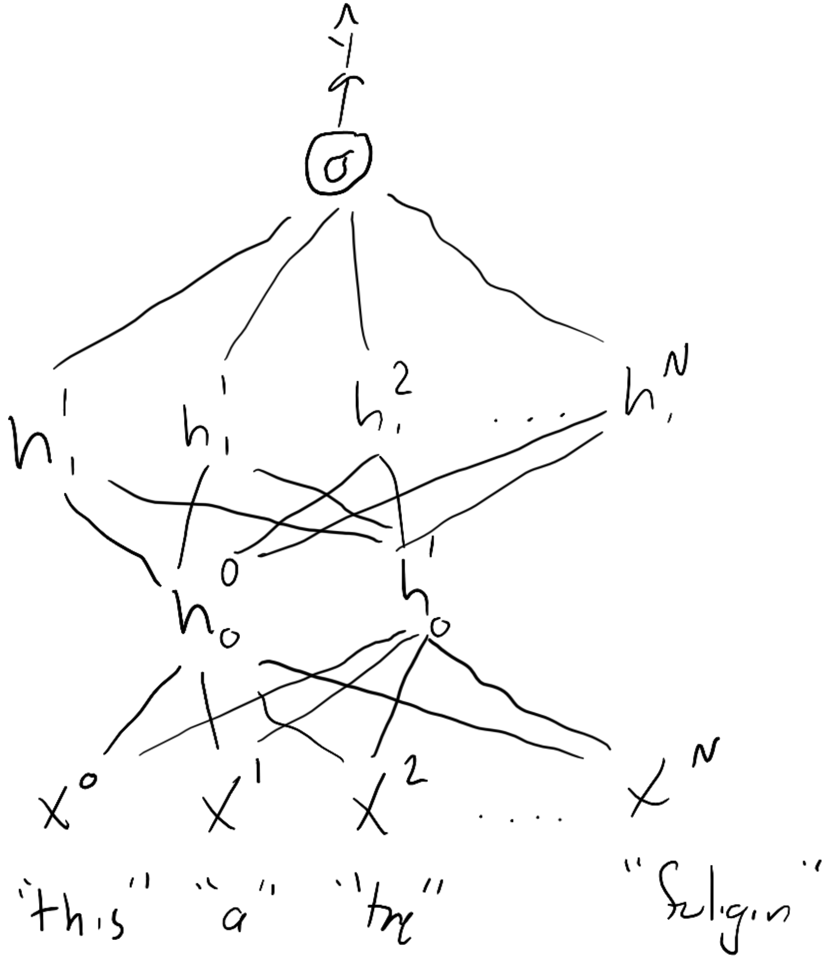
Feedforward neural nets

Or like this:



Feedforward neural nets

Or…



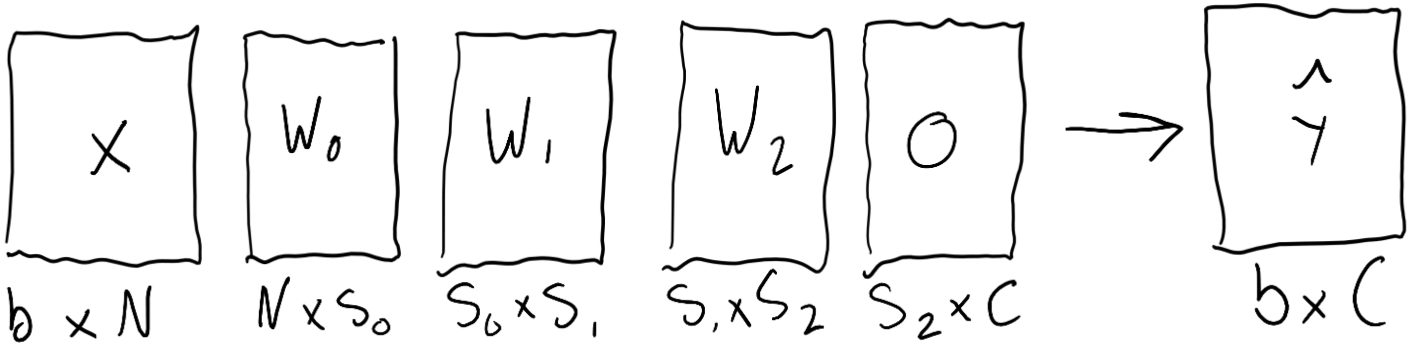
Feed-forward neural nets

AKA “Multi-layer perception” (MLP)

Composed of multiple layers of parameters of size [input size x output size]

Original input tensor gets passed through layers one by one

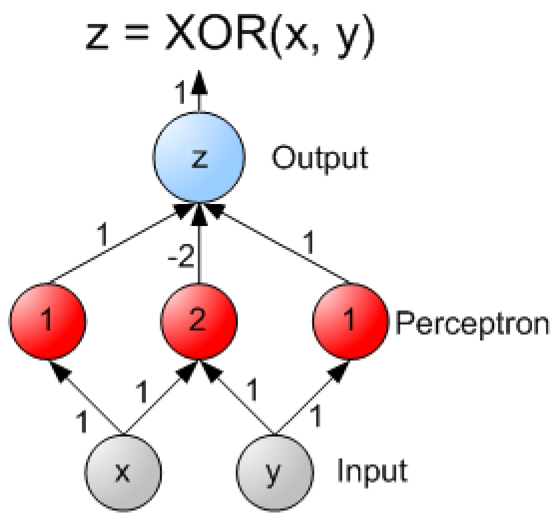
Easy to express as a series of linear algebra matrix multiplications



Why use FFNs?

By mixing and mashing the input values together, feedforward neural nets can learn more complicated functions for mapping the input X to the output $\hat{y}$

- Example: XOR logical function



https://en.wikipedia.org/wiki/Feedforward_neural_network

More generally, FFNs can model **interactions** between features

- E.g, "'Jerk' is usually predictive of toxicity, but not if the word 'chicken' is present."

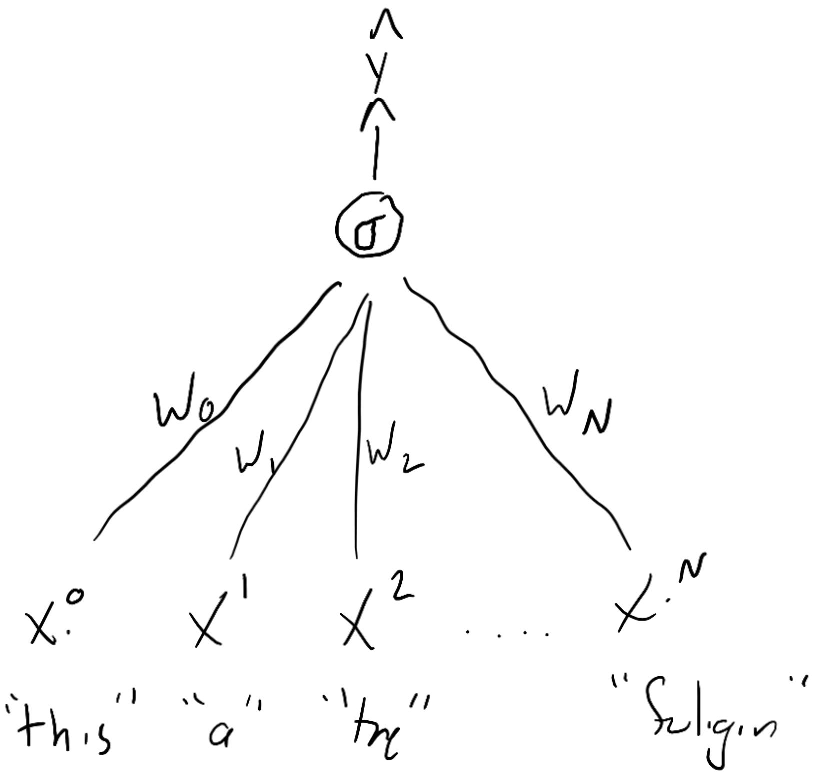
Neural nets being able to model nonlinear functions is why they outperform other methods

- If you can get the training to work

More layers is the "deep" in "deep learning"

Gradients for FFNs

It's relatively straightforward to calculate loss-parameter gradients for linear functions, because they decompose nicely into individual pieces that we can consider one at a time

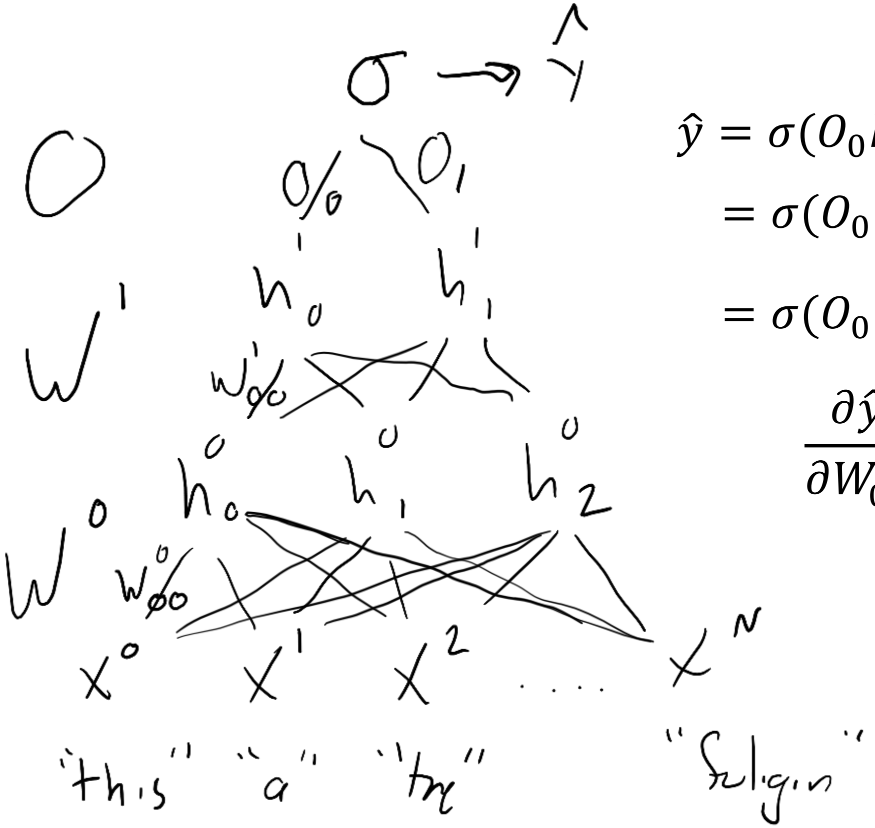


$$\hat{y} = \sigma(W_0X_0 + W_1X_1 + W_2X_2 + \cdots + W_NX_N + b)$$

$$\frac{\partial \hat{y}}{\partial W_0} = \frac{d}{\partial W_0}\,\sigma(W_0X_0)$$

Gradients for FFNs

But what about when everything now depends on everything?



$$\hat{y} = \sigma(O_0 h_0^1 + O_1 h_1^1)$$

$$= \sigma(O_0(W_{00}^1 h_0^0 + W_{10}^1 h_1^0 + W_{20}^1 h_2^0) + O_1(W_{01}^1 h_0^0 + W_{11}^1 h_1^0 + W_{21}^1 h_2^0))$$

$$= \sigma(O_0(W_{00}^1(W_{00}^0 x^0 + W_{10}^0 x^1 + W_{20}^0 x^2 + \cdots + W_{N0}^0 x^N) + \cdots) + \cdots$$

$$\frac{\partial \hat{y}}{\partial W_{00}^0} = ???$$

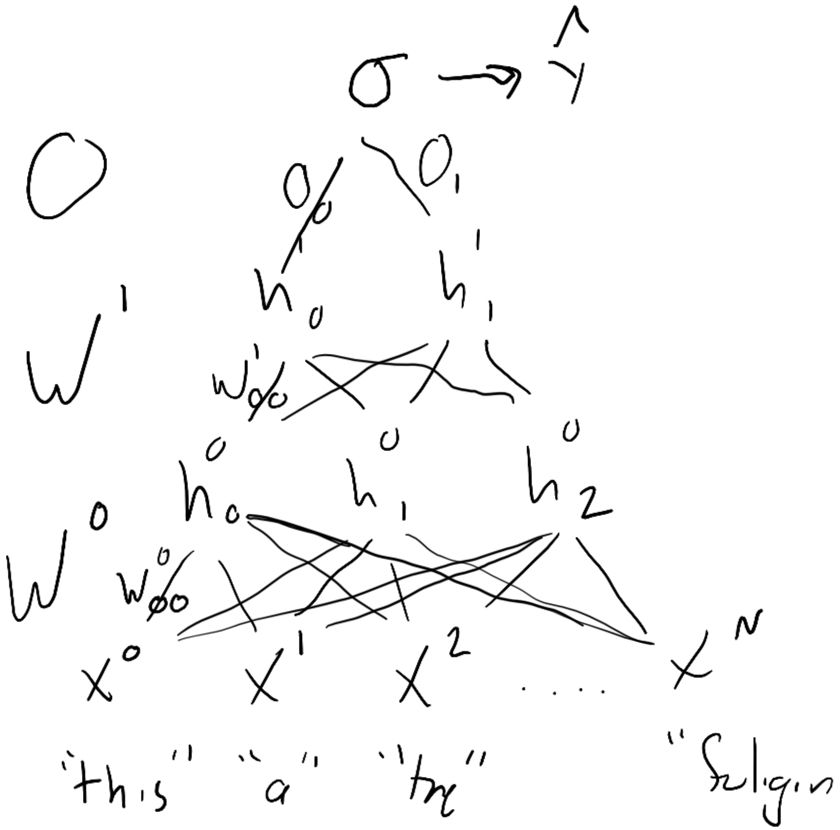
Backpropagation

Algorithm for **propagating** gradients backward from the end of a neural net to the beginning

Makes use of the chain rule: $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$

$$\frac{\partial \hat{y}}{\partial W_{00}^{0}} = \frac{\partial \hat{y}}{\partial O_0}\frac{\partial O_0}{\partial W_{00}^{0}} + \frac{\partial \hat{y}}{\partial O_0}\frac{\partial O_0}{\partial W_{00}^{0}}$$

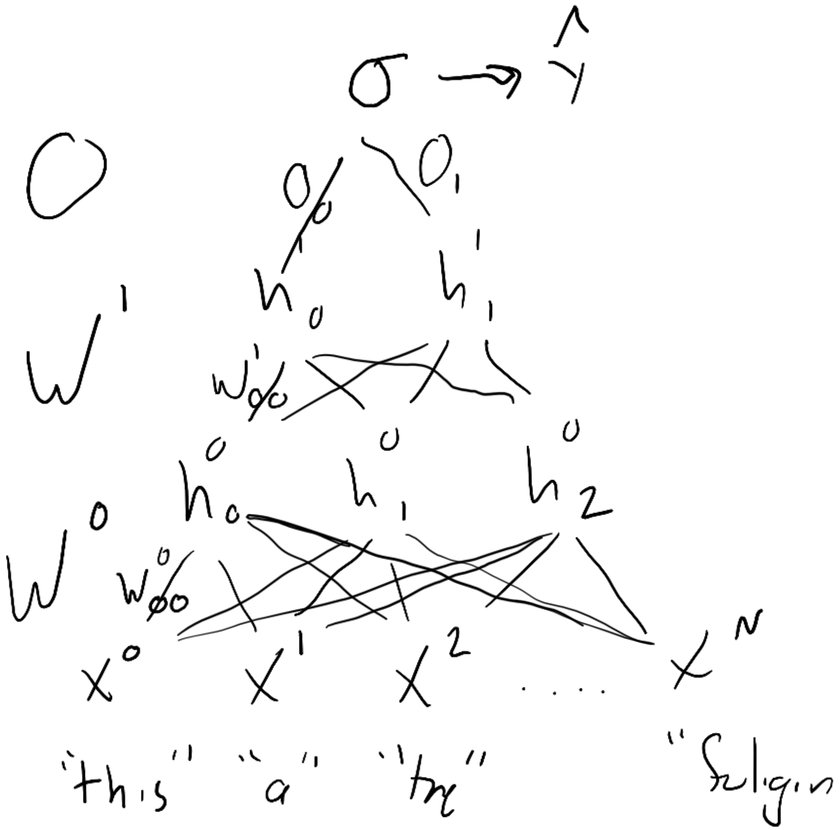
$$= \frac{\partial \hat{y}}{\partial O_0}\left(\frac{\partial O_0}{\partial W_{00}^{1}}\frac{\partial W_{00}^{1}}{\partial W_{00}^{0}} + \frac{\partial O_0}{\partial W_{10}^{1}}\frac{\partial W_{10}^{1}}{\partial W_{00}^{0}} + \frac{\partial O_0}{\partial W_{20}^{1}}\frac{\partial W_{20}^{1}}{\partial W_{00}^{0}}\right) + \cdots$$



Backpropagation

Key things to remember:

- Feedforward neural nets become math spaghetti… but they are still ultimately differentiable
- Backpropagation traces the spaghetti from the top to the bottom to figure out $\frac{\partial \hat{y}}{\partial w}$ for any arbitrary parameter w
- Pytorch does all the heavy lifting for you when you call loss.backward()
- **BUT:** the deeper down the parameter, the weaker the gradients are
  - So training tends to hit top-level layers harder than bottom-level layers



Auto-differentiation in PyTorch

PyTorch implements backpropagation by:

- Tracking layer-to-layer gradients as operations are performed in the neural net
- Applying backpropagation algorithm to obtain layer-to-loss gradients when you call loss.backward()

And these gradients get stored in GPU memory!!!!!!

- Major source of memory leaks in PyTorch

This is why it is important to:

- Wrap PyTorch operations in with torch.no_grad() when you aren't going to do training
- Zero the existing gradients before each training step

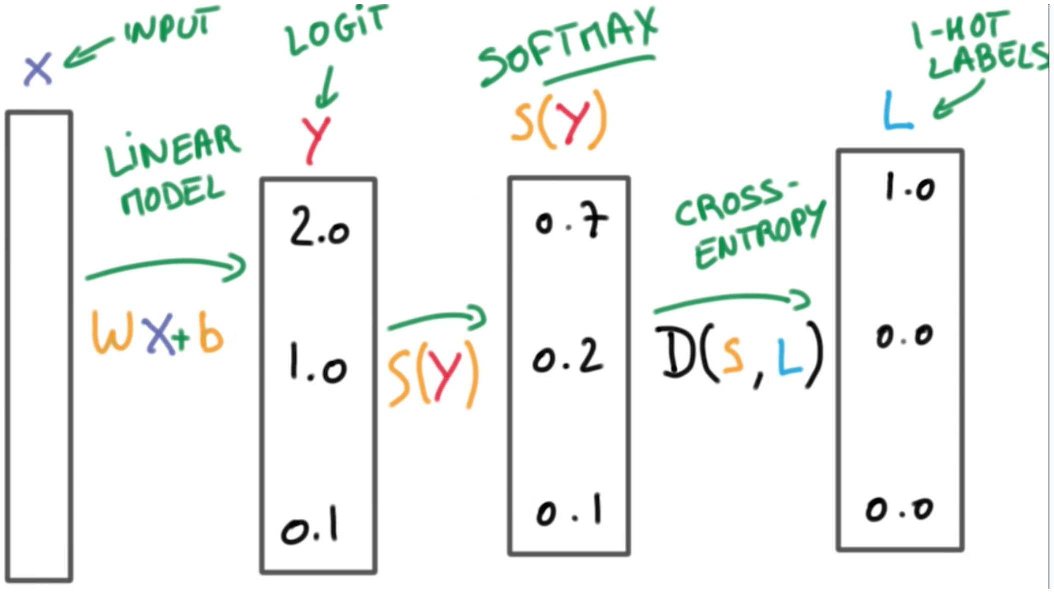
Logits and softmax

Prior to now, I've demonstrated **binary** models that spit out a single scalar logit, which is then passed through a logistic function to be squeezed to between 0 and 1

More typical is for the final layer of model (called the **output layer**) to spit out a **vector** of logits, one for each possible class, which then get passed through a **softmax** function so that they sum to one.

- So each final output value represents the probability of that class

Example of logits for logistic regression



https://www.ritchieng.com/machine-learning/deep-learning/neural-nets/

GPU operations and feedforward neural nets

Code description

- Reading and preprocessing SST-2 as usual
- Creating PyTorch Dataset and DataLoader for SST-2
- Demonstration of GPU operations
- Architecture of feedforward neural net
- Manual training of the new model

Notebook headings

Reading and preprocessing SST-2 dataset

Dataset and DataLoader

GPU operations

Feedforward model

Manual training loop with GPU

Pytorch Lightning

My screwup with optimizer.zero_grad()—unintentional lesson on the dangers of writing your own training loop

Pytorch Lightning: prefabricated training loops for PyTorch models

Requires slightly more complicated model code, but makes training loop one line

Two key elements:

- **LightningModule** – all models have to extend this
- **Trainer** – used to run the training loop

LightningModule

Subclass of torch.nn.Module

Includes:

- __init__(): defines structure
- forward(): passes input through model to make output
- Trainer hooks: get called by the Trainer object at different points in the training
  - configure_optimizers(): initializes optimizer(s)
  - training_step(): calculates training loss and returns it to Trainer
  - train_epoch_end(): called at end of training epoch for e.g. calculating accuracy
  - validation_step(): calculates validation loss and returns it to Trainer
  - validation_epoch_end(): called at end of validation epoch
  - …and tons more: https://pytorch-lightning.readthedocs.io/en/stable/starter/introduction.html

PyTorch Lightning models

Code description

- Installing a needed Python package
- Demonstration of how to write a PyTorch Lightning-compatible model

Notebook headings

Pytorch Lightning

LightningModule model

Trainer

Pytorch Lightning Trainer is an object that takes in a LightningModule and a couple of PyTorch DataLoaders (train and validation), and trains the LightningModule

Hugely powerful, tons of functionality:

- Early stopping
- Logging
- Different dev set evaluation intervals (every 0.25 epochs, every 500 steps, etc.)
- GPU vs CPU
- …and so on. You definitely want to check out the docs if you are going to use PL

https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html

PyTorch Lightning training

Code description

- Demonstration of the creation and use of a PyTorch Lightning Trainer

Notebook headings

Trainer

Concluding thoughts

New concepts

- Feedforward neural nets
  - Concept of a “layer” of a neural net architecture
- Backpropagation
- GPU operations on tensors
- Training on GPU
- Pytorch Lightning
  - LightningModule
  - Trainer