Feedforward neural nets and Pytorch Lightning

CS 780/880 Natural Language Processing Lecture 13

Samuel Carton, University of New Hampshire

Last lecture

PyTorch: Machine learning Legos

Mini-batch gradient descent

- Batch size very important

Training loop

- I screwed up!!
- Important to optimizer.zero_grad() on every training step

Avoid overfitting by:

- Regularization
- Early stopping

Training loop

```
for epoch_num in range(num_epochs):

  print(f'\nEpoch {epoch_num}')
  train_losses = []
  train_pys = []
  train_ys = []
  for step_num, train_batch in enumerate(train_dataloader):
    optimizer.zero_grad()
    train_output = our_model(**train_batch)
    train_loss = train_output['loss']
    if step_num >0 and step_num % 500 == 0: print(f'\tStep {step_num} mean training loss: {np.mean(train_losses[-500:]):.3f}')
    train_losses.append(train_loss.detach().numpy())
    train_ys.append(train_batch['y'].detach().numpy())
    train_pys.append(train_output['py'].detach().numpy())
    train_loss.backward()
    optimizer.step()

  print(f'Epoch mean train loss: {np.mean(train_losses):.3f}')
  print(f'Epoch train accuracy:{accuracy_score(np.concatenate(train_ys), np.concatenate(train_pys)):.3f}')

  dev_pys = []
  dev_ys = []
  for dev_batch in dev_dataloader:
    with torch.no_grad():
      dev_output = our_model(**dev_batch)
    dev_ys.append(dev_batch['y'].detach().numpy())
    dev_pys.append(dev_output['py'].detach().numpy())

  print(f'Epoch dev accuracy:{accuracy_score(np.concatenate(dev_ys), np.concatenate(dev_pys)):.3f}')
```

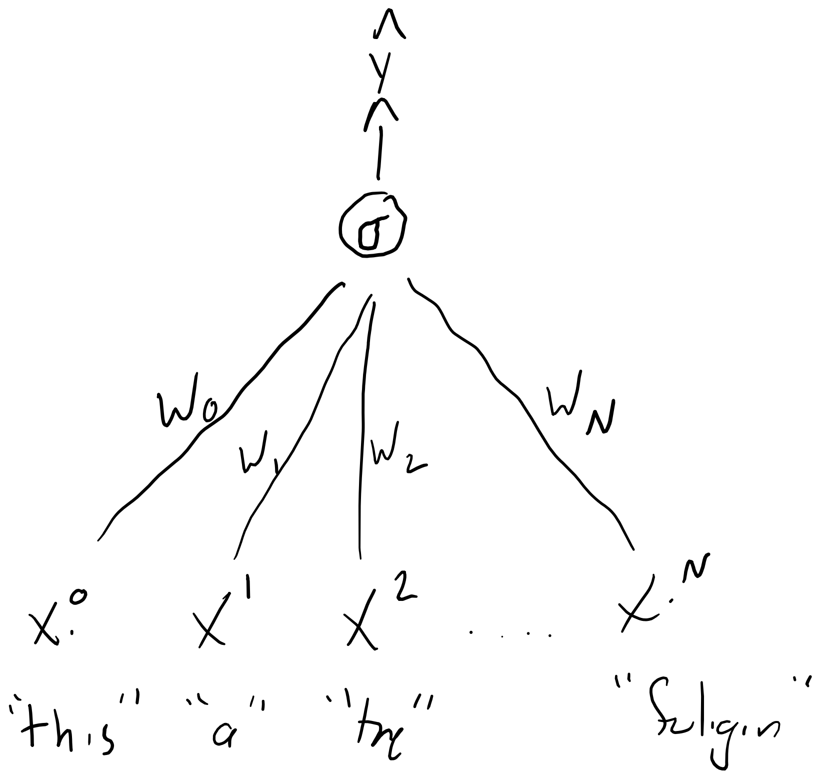
Training loop

```
Epoch 0
        Step 500 mean training loss: 0.561
        Step 1000 mean training loss: 0.438
        Step 1500 mean training loss: 0.389
        Step 2000 mean training loss: 0.359
        Step 2500 mean training loss: 0.338
        Step 3000 mean training loss: 0.321
        Step 3500 mean training loss: 0.307
        Step 4000 mean training loss: 0.308
Epoch mean train loss: 0.374
Epoch train accuracy:0.850
Epoch dev accuracy:0.817

Epoch 1
        Step 500 mean training loss: 0.256
        Step 1000 mean training loss: 0.259
        Step 1500 mean training loss: 0.266
        Step 2000 mean training loss: 0.256
        Step 2500 mean training loss: 0.259
        Step 3000 mean training loss: 0.253
        Step 3500 mean training loss: 0.254
        Step 4000 mean training loss: 0.255
Epoch mean train loss: 0.257
Epoch train accuracy:0.903
Epoch dev accuracy:0.822
```

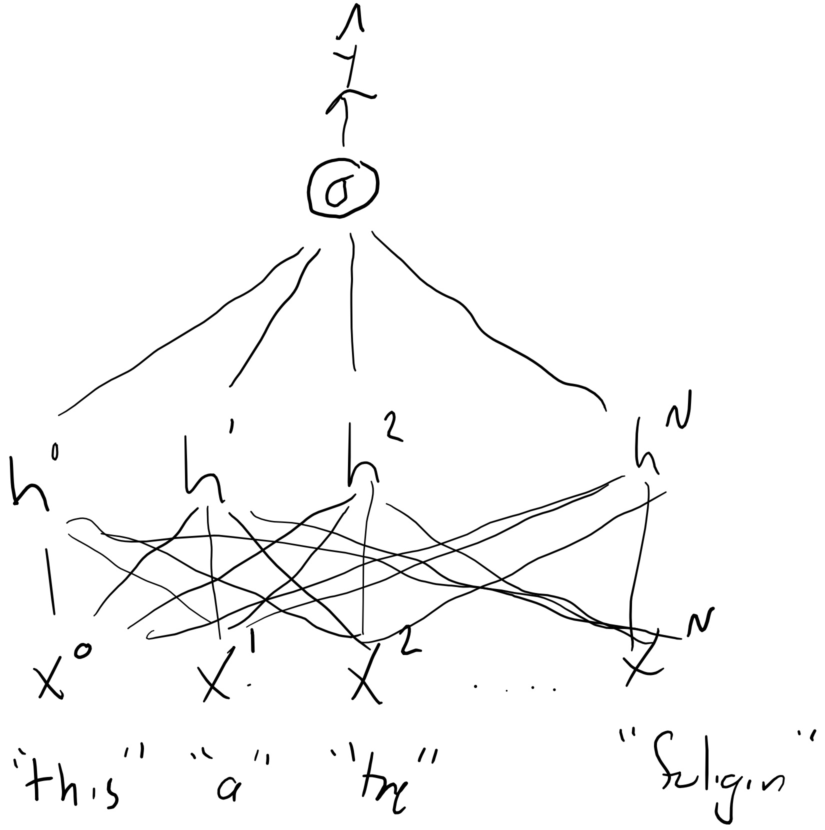
Feedforward neural nets

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



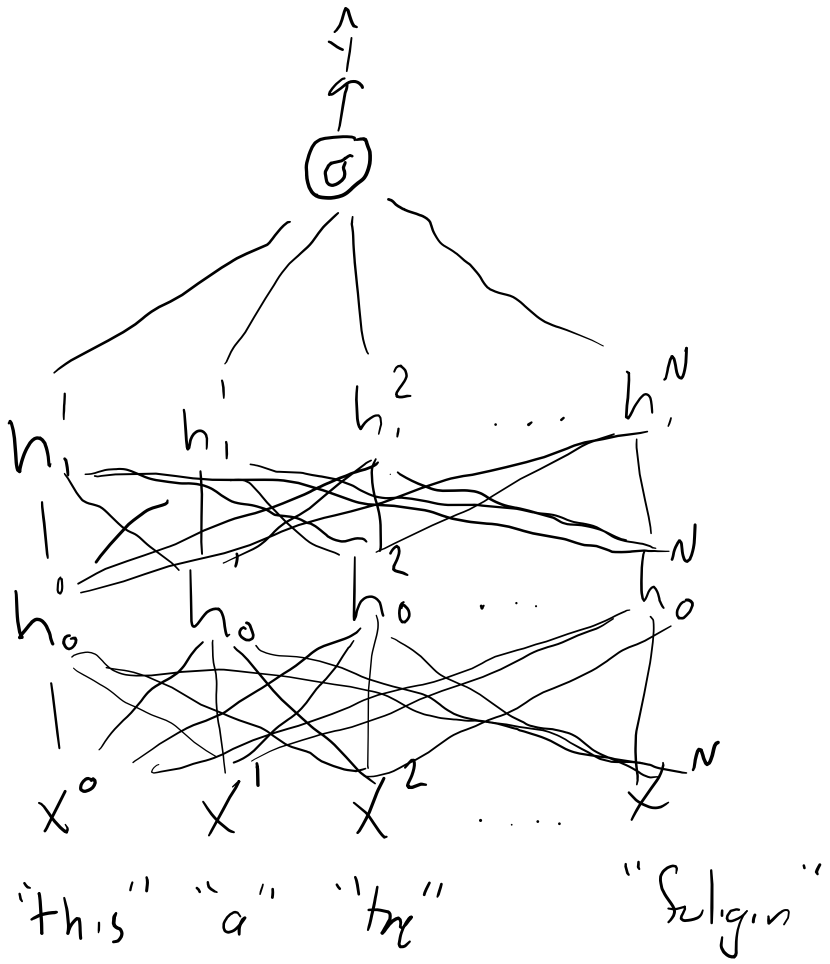
Feedforward neural nets

But what about models that look like this:



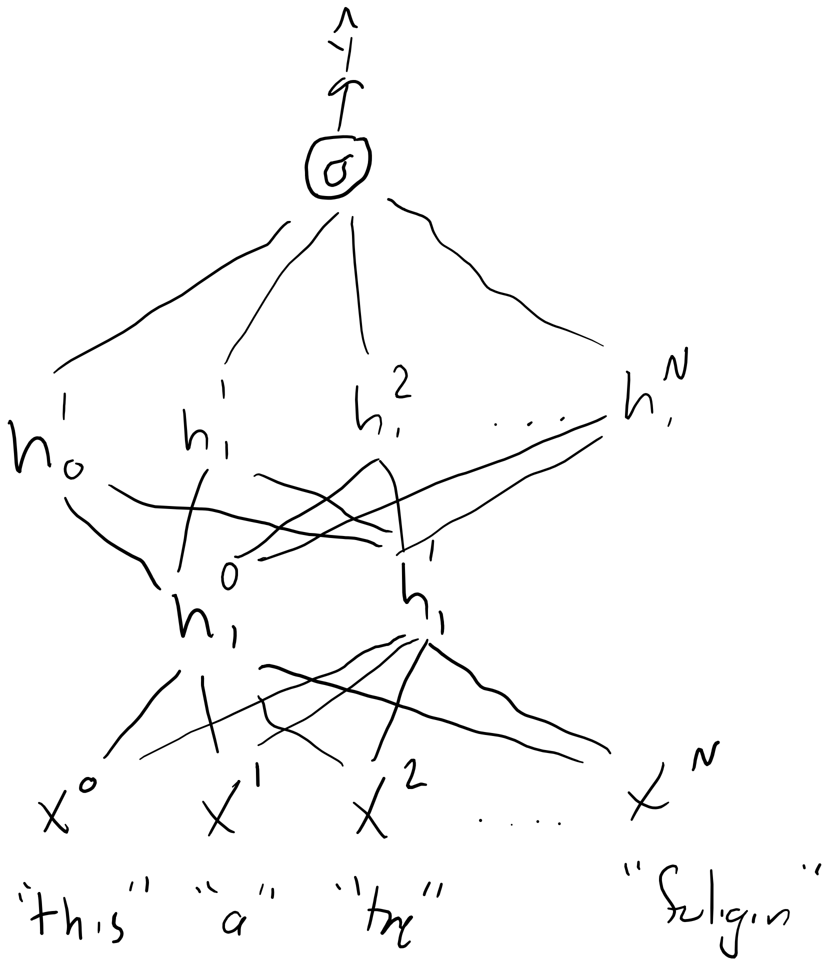
Feedforward neural nets

Or like this:



Feedforward neural nets

Or…



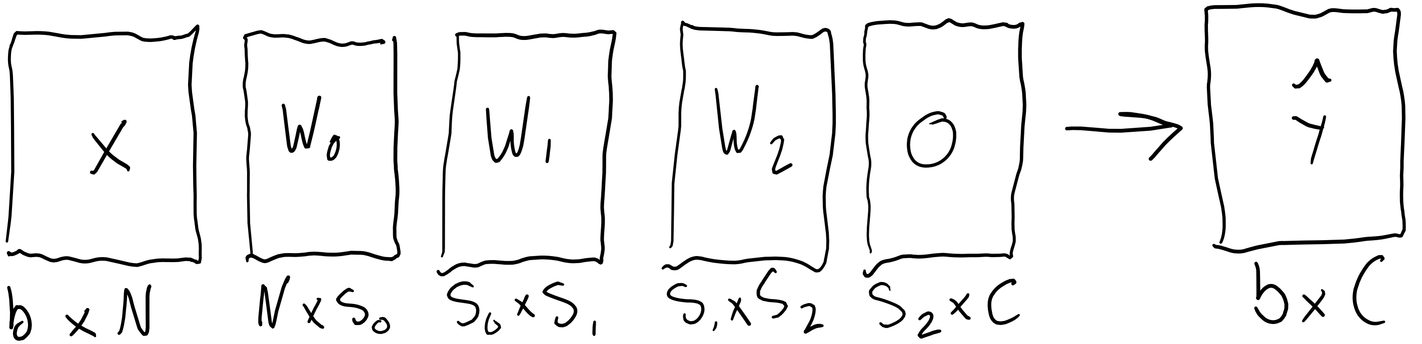
Feed-forward neural nets

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

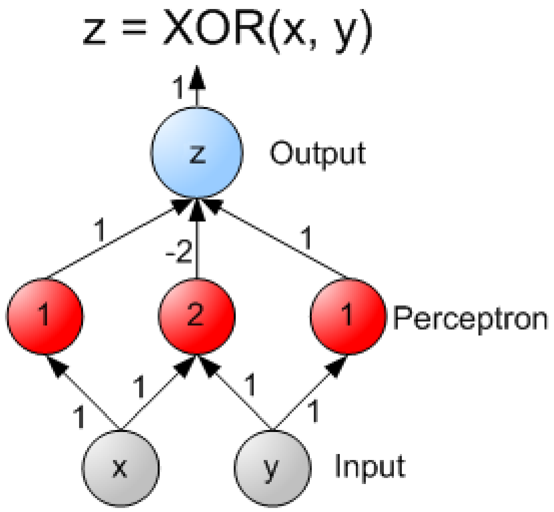
More generally, FFNs can model **interactions** between features

- 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

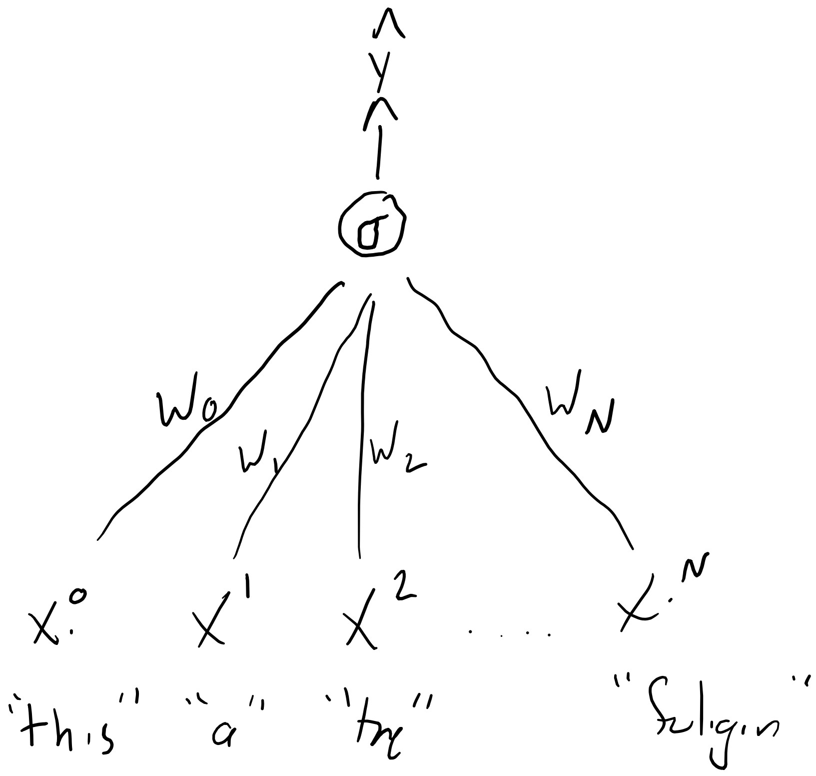
More layers is the "deep" in "deep learning"



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

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

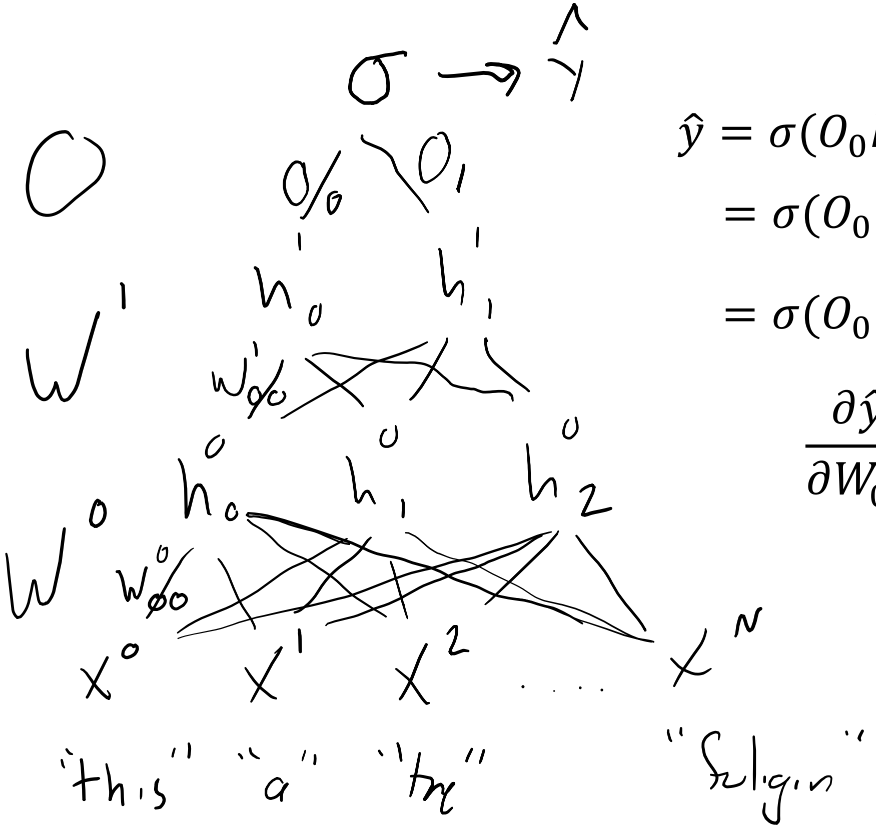


$$\hat{y} = \sigma(W_0X_0 + W_1X_1 + W_2X_2 + \cdots + W_NX_N + b)$$

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

Gradients for FFNs

But what about when everything now depends on everything?



$$\hat{y} = \sigma(O_0 h_0^1 + O_1 h_1^1)$$
$$= \sigma(O_0(W_{00}^1 h_0^0 + W_{10}^1 h_1^0 + W_{20}^1 h_2^0) + O_1(W_{01}^1 h_0^0 + W_{11}^1 h_1^0 + W_{21}^1 h_2^0))$$
$$= \sigma(O_0(W_{00}^1(W_{00}^0 x^0 + W_{10}^0 x^1 + W_{20}^0 x^2 + \cdots + W_{N0}^0 x^N) + \cdots) + \cdots$$

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

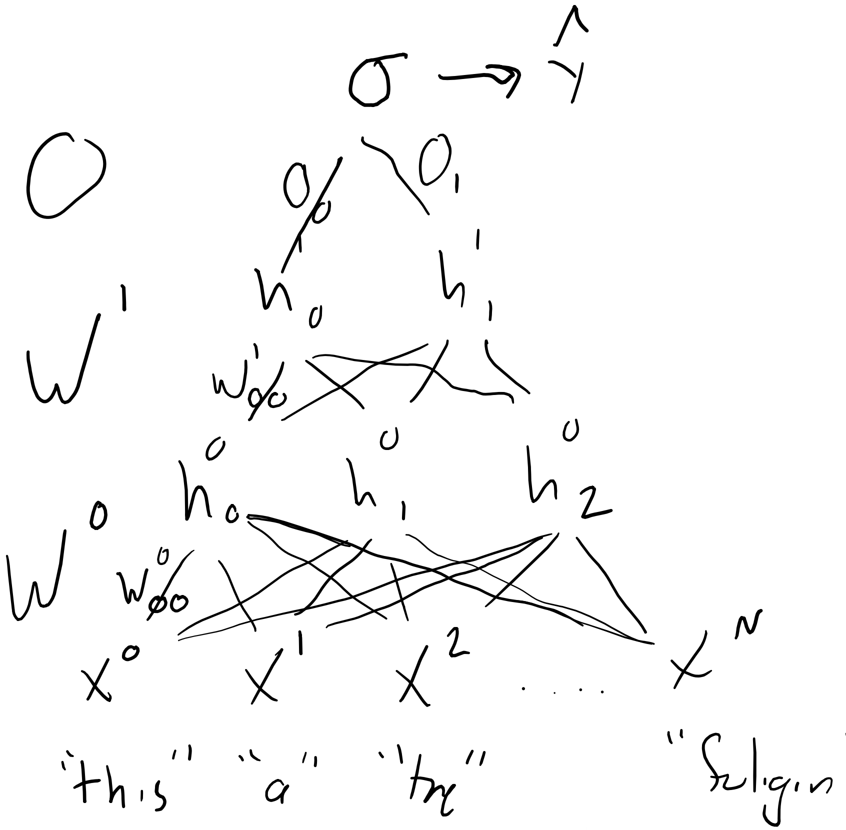
Backpropagation

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

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

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

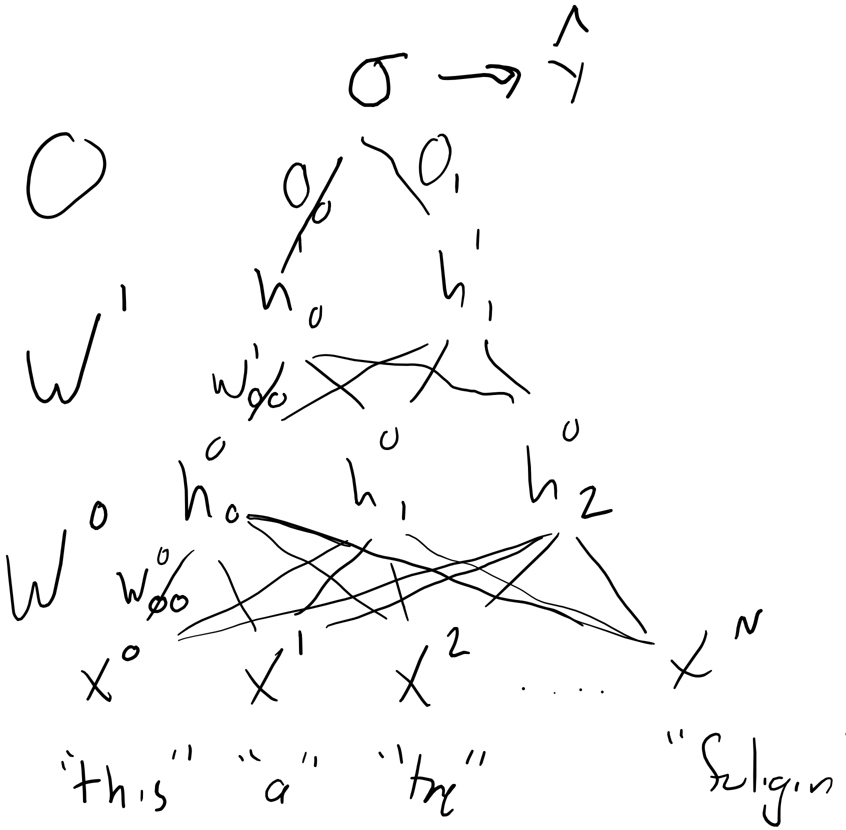
$$= \frac{\partial \hat{y}}{\partial O_0} \left(\frac{\partial O_0}{\partial W_{00}^{1}} \frac{\partial W_{00}^{1}}{\partial W_{00}^{0}} + \frac{\partial O_0}{\partial W_{10}^{1}} \frac{\partial W_{10}^{1}}{\partial W_{00}^{0}} + \frac{\partial O_0}{\partial W_{20}^{1}} \frac{\partial W_{20}^{1}}{\partial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



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



GPU operations

```
# This is how you check if there's a GPU available:
# You can make one available in Runtime --> Change runtime type --> Hardware accelerator --> GPU
torch.cuda.is_available()
```

```
True
```

```
# This defines a random 1000x1000 tensor
t = torch.rand((1000,1000))

display(t)

# This switches the tensor onto the GPU if it is available
if torch.cuda.is_available():
  gt = t.cuda()
  display(gt)
```

```
tensor([[0.6234, 0.5047, 0.8788,  ..., 0.4923, 0.4688, 0.2012],
        [0.4593, 0.3015, 0.0172,  ..., 0.3370, 0.2682, 0.3077],
        [0.7464, 0.6963, 0.3865,  ..., 0.9686, 0.2653, 0.5761],
        ...,
        [0.6183, 0.4092, 0.8795,  ..., 0.1698, 0.3077, 0.8897],
        [0.1478, 0.9971, 0.4989,  ..., 0.6327, 0.9928, 0.5822],
        [0.9916, 0.4715, 0.0476,  ..., 0.8947, 0.3597, 0.3564]])
tensor([[0.6234, 0.5047, 0.8788,  ..., 0.4923, 0.4688, 0.2012],
        [0.4593, 0.3015, 0.0172,  ..., 0.3370, 0.2682, 0.3077],
        [0.7464, 0.6963, 0.3865,  ..., 0.9686, 0.2653, 0.5761],
        ...,
        [0.6183, 0.4092, 0.8795,  ..., 0.1698, 0.3077, 0.8897],
        [0.1478, 0.9971, 0.4989,  ..., 0.6327, 0.9928, 0.5822],
        [0.9916, 0.4715, 0.0476,  ..., 0.8947, 0.3597, 0.3564]],
       device='cuda:0')
```

GPU operations

```
# You can't convert a tensor back to numpy until you move it back to CPU
try:
  n_gt = gt.numpy()
except Exception as ex:
  print(ex)
```

```
can't convert cuda:0 device type tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory first.
```

```
# And you can't perform operations between tensors on different devices
try:
  m_t = t* gt
except Exception as ex:
  print(ex)
```

```
Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!
```

GPU operations

```
# We can see the effect of GPU if we do a little benchmarking, by repeating
# an expensive operation over and over again
starttime = datetime.now()
calc_t = t
for i in range(300):
  calc_t = torch.nn.functional.normalize(torch.matmul(calc_t, calc_t),dim=1)
endtime = datetime.now()
print('Elapsed time:',endtime-starttime)
display(calc_t)
```

```
Elapsed time: 0:00:09.207511
tensor([[0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307],
        [0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307],
        [0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307],
        ...,
        [0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307],
        [0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307],
        [0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307]])
```

```
# We can see the savings when we do this via GPU
if torch.cuda.is_available():
  starttime = datetime.now()
  calc_t = t.cuda()
  for i in range(300):
    calc_t = torch.nn.functional.normalize(torch.matmul(calc_t, calc_t),dim=1)
  endtime = datetime.now()
  print('Elapsed time:',endtime-starttime)
  display(calc_t)
```

```
Elapsed time: 0:00:00.082512
tensor([[0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307],
        [0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307],
        [0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307],
        ...,
        [0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307],
        [0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307],
        [0.0320, 0.0317, 0.0319,  ..., 0.0313, 0.0319, 0.0307]],
       device='cuda:0')
```

Feedforward model

```
class FeedForwardClassifier(torch.nn.Module):
    def __init__(self,
                 vocab_size:int,
                 num_classes:int=2):
        super(FeedForwardClassifier, self).__init__()
        self.layer_0 = torch.nn.Linear(vocab_size, 100)
        self.layer_1 = torch.nn.Linear(100, 100)
        self.layer_2 = torch.nn.Linear(100, 100)
        self.output_layer = torch.nn.Linear(100, num_classes, bias=True)
        self.softmax = torch.nn.Softmax(dim=1)

    def forward(self, X:torch.Tensor, y:torch.Tensor):
        intermediate_0 = self.layer_0(X) #(batch size, 100)
        intermediate_1 = self.layer_1(intermediate_0) #(batch size, 100)
        intermediate_2 = self.layer_2(intermediate_1) # (batch size, 100)
        py_logits = self.output_layer(intermediate_2) # (batch size, num_classes)
        py_probs = self.softmax(py_logits) # (batch size, num_classes)
        py = torch.argmax(py_probs, dim=1)
        loss = torch.nn.functional.cross_entropy(py_logits, y, reduction ='mean')
        return {'py_logits':py_logits,
                'py_probs':py_probs,
                'py':py,
                'loss':loss}
```

Feedforward model

```
# Again we can instantiate the model and look at it
vocab_size = train_X.shape[1] # We need to know this in order  to set up the model
our_model = FeedForwardClassifier(vocab_size=vocab_size, num_classes=2)
# Displaying the model will show its layers
display(our_model)
```

```
FeedForwardClassifier(
  (layer_0): Linear(in_features=10106, out_features=100, bias=True)
  (layer_1): Linear(in_features=100, out_features=100, bias=True)
  (layer_2): Linear(in_features=100, out_features=100, bias=True)
  (output_layer): Linear(in_features=100, out_features=2, bias=True)
  (softmax): Softmax(dim=1)
)
```

Manual training loop with GPU

```
learning_rate = 0.001

# Scoot the model onto GPU
our_model.cuda() # This operation is in-place for PyTorch Modules but not for tensors

# It was important to do that before making the optimizer
optimizer = torch.optim.Adam(our_model.parameters(), lr=learning_rate)

num_epochs = 1
```

```
# I will remake the dataloaders with a bigger batch size for speed
batch_size = 32
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
dev_dataloader = torch.utils.data.DataLoader(dev_dataset, batch_size=batch_size, shuffle=False)
```

```
# And I have to define a couple helper functions to move a dictionary of tensors on/off GPU
from typing import Dict

def to_gpu(d:Dict[str, torch.Tensor]):
    return {key:d[key].cuda() for key in d}

def to_cpu(d:Dict[str, torch.Tensor]):
    return {key:d[key].cpu() for key in d}
```

Manual training loop with GPU

```python
from sklearn.metrics import accuracy_score
for epoch_num in range(num_epochs):

  print(f'\nEpoch {epoch_num}')
  train_losses = []
  train_pys = []
  train_ys = []
  for step_num, train_batch in enumerate(train_dataloader):
    optimizer.zero_grad()
    train_output = to_cpu(our_model(**to_gpu(train_batch)))
    train_loss = train_output['loss']
    if step_num >0 and step_num % 500 == 0:
      print(f'\tStep {step_num} mean training loss: {np.mean(train_losses[-500:]):.3f}')

      # Dev set evaluation
      dev_pys = []
      dev_ys = []
      for dev_batch in dev_dataloader:
        with torch.no_grad():
          dev_output = to_cpu(our_model(**to_gpu(dev_batch)))
        dev_ys.append(dev_batch['y'].detach().numpy())
        dev_pys.append(dev_output['py'].detach().numpy())
      print(f'\tDev accuracy:{accuracy_score(np.concatenate(dev_ys), np.concatenate(dev_pys)):.3f}')

    train_losses.append(train_loss.detach().numpy())
    train_ys.append(train_batch['y'].detach().numpy())
    train_pys.append(train_output['py'].detach().numpy())
    train_loss.backward()
    optimizer.step()

  print(f'Mean train loss: {np.mean(train_losses):.3f}')
  print(f'Train accuracy:{accuracy_score(np.concatenate(train_ys), np.concatenate(train_pys)):.3f}')
  print(f'Dev accuracy:{accuracy_score(np.concatenate(dev_ys), np.concatenate(dev_pys)):.3f}')
```

```
Epoch 0
	Step 500 mean training loss: 0.451
	Dev accuracy:0.796
	Step 1000 mean training loss: 0.353
	Dev accuracy:0.807
	Step 1500 mean training loss: 0.310
	Dev accuracy:0.805
	Step 2000 mean training loss: 0.303
	Dev accuracy:0.804
Mean train loss: 0.351
Train accuracy:0.854
Dev accuracy:0.804

Epoch 1
	Step 500 mean training loss: 0.243
	Dev accuracy:0.797
	Step 1000 mean training loss: 0.262
	Dev accuracy:0.803
	Step 1500 mean training loss: 0.259
	Dev accuracy:0.800
	Step 2000 mean training loss: 0.244
	Dev accuracy:0.800
Mean train loss: 0.253
Train accuracy:0.904
Dev accuracy:0.800
```

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

Pytorch Lightning

```
# This is the first time you will have had to install a package that doesn't come standard in Google Colab

# It's easy:
! pip install --quiet "pytorch-lightning"
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 827.8/827.8 KB 14.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 518.6/518.6 KB 42.3 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.0/1.0 MB 59.7 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 158.8/158.8 KB 13.9 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 199.2/199.2 KB 22.7 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 264.6/264.6 KB 28.4 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 114.2/114.2 KB 12.6 MB/s eta 0:00:00
```

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

LightningModule model

```python
class PLFeedForwardClassifier(pl.LightningModule):
  def __init__(self,
              vocab_size:int,
              num_classes:int=2,
              learning_rate:float=1e-3,
              **kwargs):
    super().__init__( **kwargs)
    self.layer_0 = torch.nn.Linear(vocab_size, 100)
    self.layer_1 = torch.nn.Linear(100, 100)
    self.layer_2 = torch.nn.Linear(100, 100)
    self.layers=[self.layer_0, self.layer_1, self.layer_2]
    self.output_layer = torch.nn.Linear(100, num_classes, bias=True)
    self.learning_rate = learning_rate
    self.train_accuracy = Accuracy(task='binary')
    self.val_accuracy = Accuracy(task='binary')

  def forward(self, X:torch.Tensor, y:torch.Tensor):
    intermediate = X
    for layer in self.layers:
      intermediate= layer(intermediate)
    py_logits = self.output_layer(intermediate)
    py_probs = torch.nn.functional.softmax(py_logits, dim=1)
    py = torch.argmax(py_logits, dim=1)
    loss = torch.nn.functional.cross_entropy(py_logits, y, reduction='mean')
    return {
        'py_probs':py_probs,
        'py':py,
        'loss':loss
    }
```

LightningModule model

```python
def configure_optimizers(self):
  return [torch.optim.Adam(self.parameters(), lr=self.learning_rate)]

def training_step(self, batch, batch_idx):
  result = self.forward(**batch)
  loss = result['loss']
  self.log('train_loss', result['loss'])
  self.train_accuracy.update(result['py'], batch['y'])
  return loss

def training_epoch_end(self, outs):
  self.log('train_accuracy', self.train_accuracy)
  print('Training accuracy:', self.train_accuracy.compute())

def validation_step(self, batch, batch_idx):
  result = self.forward(**batch)
  self.val_accuracy(result['py'], batch['y'])
  return result['loss']

def validation_epoch_end(self, outs):
  self.log('val_accuracy', self.val_accuracy)
  print('Validation accuracy:', self.val_accuracy.compute())
```

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

Trainer

```python
# Instantiating the model works just the same way
# except we're giving the learning rate directly to the model now
pl_model = PLFeedForwardClassifier(vocab_size=vocab_size,
                                   num_classes = 2,
                                   learning_rate = 0.001)
```

```python
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks.progress import TQDMProgressBar
```

```python
# We define a Trainer we will use to train our model
# There are a TON of options you can set:
# https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html
trainer = Trainer(
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None,
    max_epochs=1,
    callbacks=[TQDMProgressBar(refresh_rate=20)],
    val_check_interval = 0.25,
    )

```

Trainer

```
# And then actually fitting the model is one line
trainer.fit(model=pl_model,
            train_dataloaders=train_dataloader,
            val_dataloaders=dev_dataloader) # I know I'm being inconsistent with val vs. dev
```

```
WARNING:pytorch_lightning.loggers.tensorboard:Missing logger folder: /content/lightning_logs
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name           | Type           | Params
--------------------------------------------------
0 | layer_0        | Linear         | 1.0 M
1 | layer_1        | Linear         | 10.1 K
2 | layer_2        | Linear         | 10.1 K
3 | output_layer   | Linear         | 202
4 | train_accuracy | BinaryAccuracy | 0
5 | val_accuracy   | BinaryAccuracy | 0
--------------------------------------------------
1.0 M     Trainable params
0         Non-trainable params
1.0 M     Total params
4.124     Total estimated model params size (MB)
Validation accuracy: tensor(0.4844, device='cuda:0')
```

Epoch 0: 100% 2217/2217 [00:34<00:00, 63.44it/s, loss=0.32, v_num=0]

```
Validation accuracy: tensor(0.8050, device='cuda:0')
Validation accuracy: tensor(0.8016, device='cuda:0')
Validation accuracy: tensor(0.8016, device='cuda:0')
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=1` reached.
Validation accuracy: tensor(0.8096, device='cuda:0')
Training accuracy: tensor(0.8567, device='cuda:0')
```

Concluding thoughts

Feedforward neural nets

Backpropagation

GPU operations on tensors

Training on GPU

Pytorch Lightning

- LightningModule
- Trainer