Neural Net Training with PyTorch

CS 780/880 Natural Language Processing Lecture 11

Samuel Carton, University of New Hampshire

Last lecture

Linear regression

- Learn $Wx + b$ from data
- Predict continuous values
- Optimize mean squared error

Logistic regression

- Learn $\sigma(Wx + b)$ from data
- Predict (close to) 0 or 1
- Optimize cross-entropy

Key concepts:

- Loss function
    - I.e. objective function
- Gradient of loss with respect to parameters
- Gradient descent
- Activation function

PyTorch

PyTorch is a **deep learning library**

- Define the structure of a neural net
- Use gradient descent to train it
- Implementations of common structural elements

PyTorch

- Created and maintained by Meta
- Competes primarily with TensorFlow (Google)
- Fairly dominant in research right now

All deep learning libraries are basically a lego kit for **tensor** operations

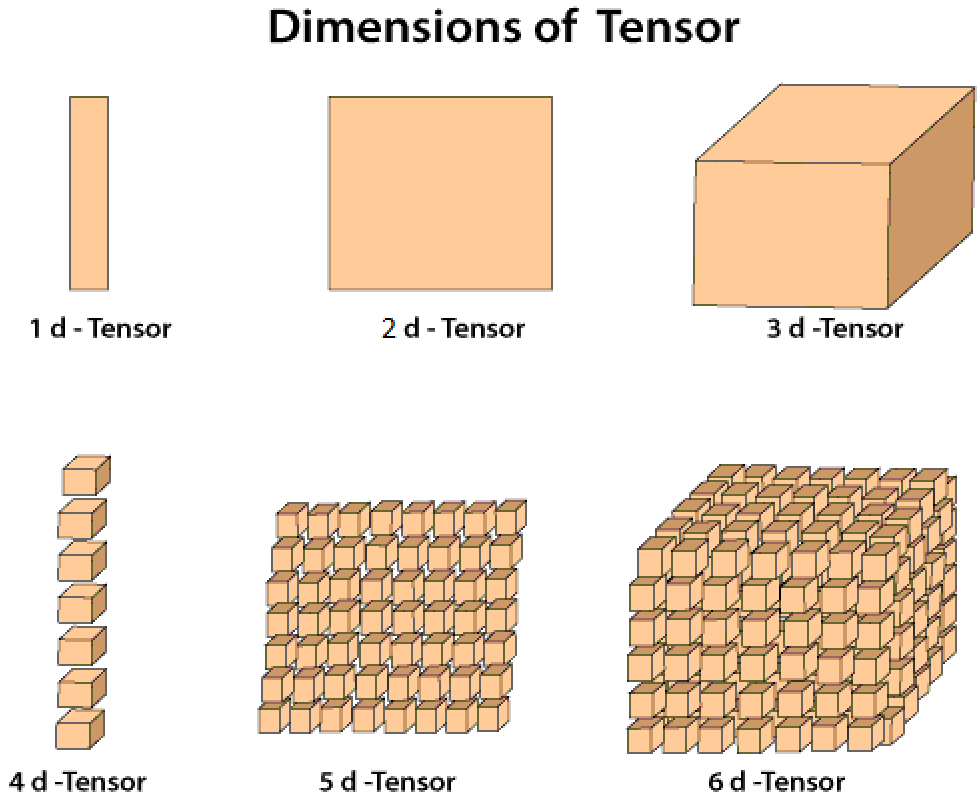
Tensors

A tensor is an N-dimensional array of values

- e.g. a scalar (0D), vector (1D), or matrix (2D)

Any neural net is basically just a bunch of tensor operations

GPUs happen to be good at doing tensor operations quickly



https://www.javatpoint.com/pytorch-tensors

Visualizing logistic regression

Recall our visualization of logistic regression as a matrix (i.e tensor) operation

$$\sigma\left(\begin{bmatrix} X_0^0 & X_0^1 & X_0^2 & \cdots & X_0^N \\ X_1^0 & X_1^1 & X_1^2 & \cdots & X_1^N \\ & & \cdots & & \\ X_M^0 & X_M^1 & X_M^2 & \cdots & X_M^N \end{bmatrix} \begin{bmatrix} W_0 \\ W_1 \\ \vdots \\ W_N \end{bmatrix} + b \right) = \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \vdots \\ \hat{y}_M \end{bmatrix}$$

Tensors – Basic operations and dimensionality

PyTorch Tensors are functionally almost identical to Numpy arrays

Basic operations

```
v0 = torch.Tensor([1,2,3])
v1 = torch.Tensor([4,5,6])
```

```
v0 + v1
```

```
tensor([5., 7., 9.])
```

```
v0 - v1
```

```
tensor([-3., -3., -3.])
```

```
v0 * v1
```

```
tensor([ 4., 10., 18.])
```

```
v0 / v1
```

```
tensor([0.2500, 0.4000, 0.5000])
```

```
v0 ** 2
```

```
tensor([1., 4., 9.])
```

Dimensionality

```
# Scalar (0-dimensional tensor)
s0 = torch.Tensor(1)
s0
```

```
tensor([1.0208e+12])
```

```
# Vector (1-dimensional tensor)
v0
```

```
tensor([1., 2., 3.])
```

```
# Matrix (2-dimensional tensor)

m0 = torch.Tensor([[1,2],[3,4]])
m0
```

```
tensor([[1., 2.],
        [3., 4.]])
```

```
# 3-dimensional tensor
t0 = torch.Tensor([[[1,2],[3,4]],[[4,5],[5,6]]])
t0
```

```
tensor([[[1., 2.],
         [3., 4.]],

        [[4., 5.],
         [5., 6.]]])
```

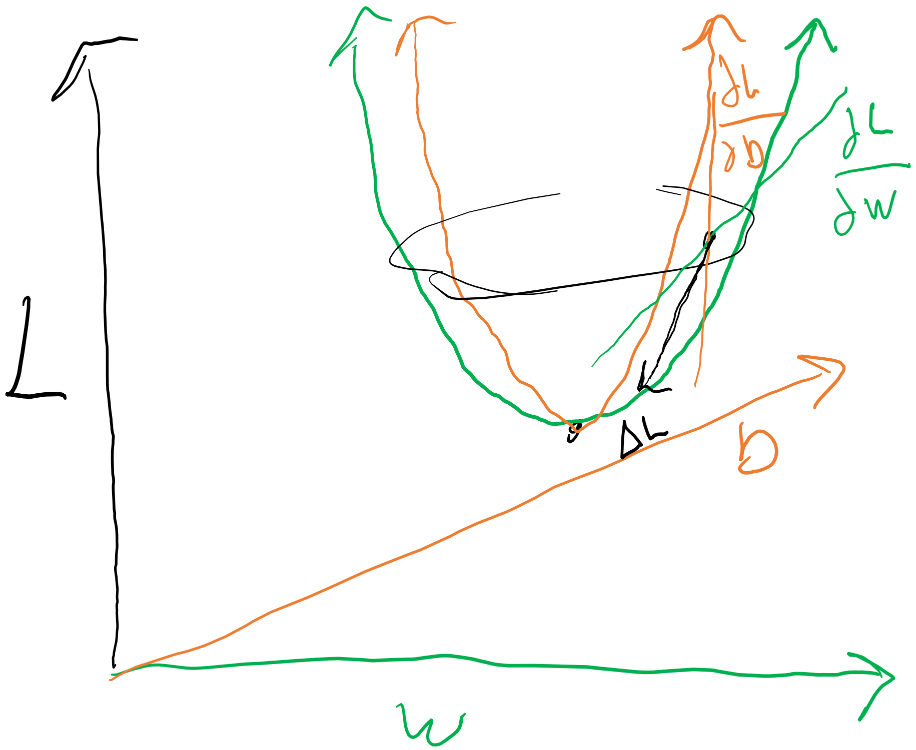
Tensors – Convenient functionality

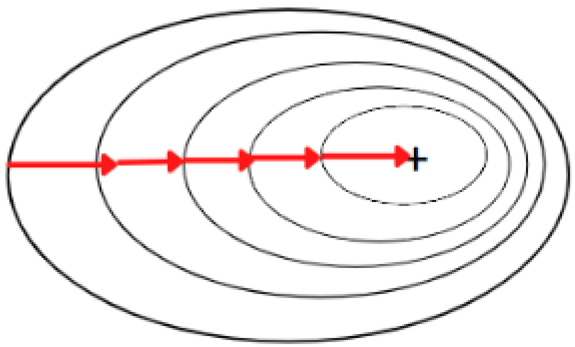
```python
# It's easy to convert back and forth between numpy arrays and tensors

print('Conversion from PyTorch to Numpy')
np_m = m0.numpy()
display(np_m)

print('\nConverstion from Numpy to PyTorch')
t_np_m = torch.Tensor(np_m)
display(t_np_m)
```

```
Conversion from PyTorch to Numpy
array([[1., 2.],
       [3., 4.]], dtype=float32)

Converstion from Numpy to PyTorch
tensor([[1., 2.],
        [3., 4.]])
```

Gradient Descent

Basic idea: Calculate the loss over the **whole training set**, do a step along the gradient, then recalculate the loss and so on





https://www.analyticsvidhya.com/blog/2022/07/gradient-descent-and-its-types/

Mini-batch gradient descent

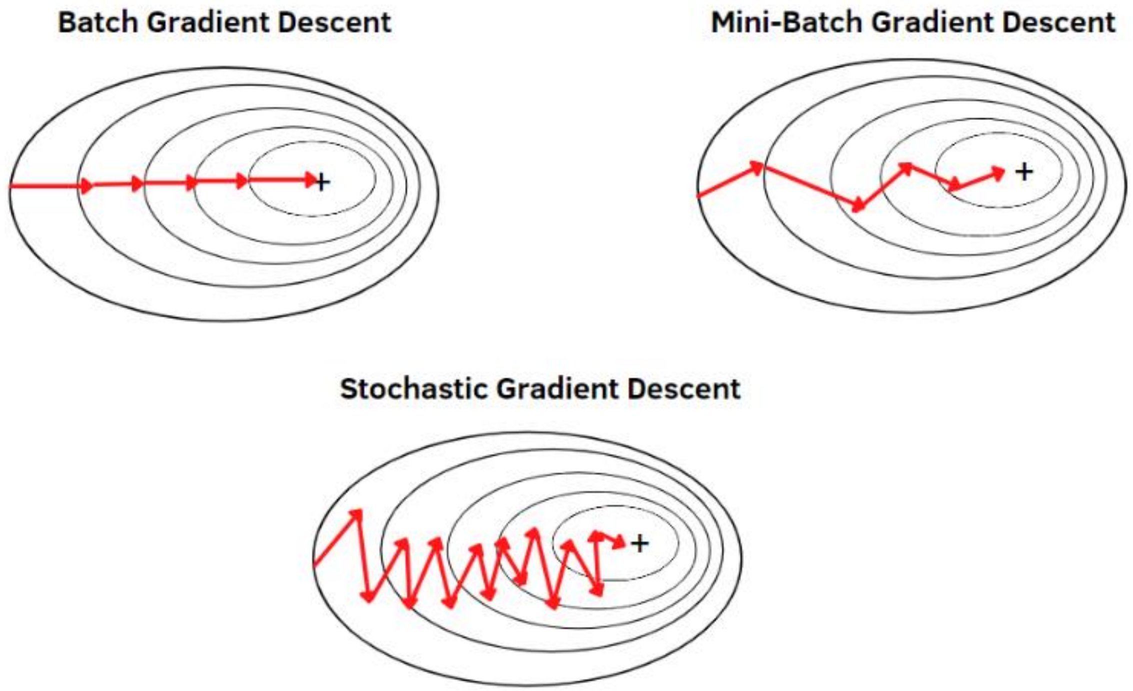
For big datasets/models, we can’t fit all training gradients in memory.

So we do our steps on **batches** of the data, one at a time

When the batch size is 1, it’s called **stochastic gradient descent**

Batch size is a **hugely important** hyperparameter in neural net training.

- Bigger usually better, but requires a bigger GPU
- Why Nvidia A100s are like $15k



Reading and preprocessing SST-2 dataset

	sentence	label	preprocessed
0	it 's a charming and often affecting journey .	1	it 's a charm and often affect journey .
1	unflinchingly bleak and desperate	0	unflinchingli bleak and desper
2	allows us to hope that nolan is poised to emba...	1	allow us to hope that nolan is pois to embark ...
3	the acting , costumes , music , cinematography...	1	the act , costum , music , cinematographi and ...
4	it 's slow -- very , very slow .	0	it 's slow -- veri , veri slow .
...	...	...	...
867	has all the depth of a wading pool .	0	ha all the depth of a wade pool .
868	a movie with a real anarchic flair .	1	a movi with a real anarch flair .
869	a subject like this should inspire reaction in...	0	a subject like thi should inspir reaction in i...
870	... is an arthritic attempt at directing by ca...	0	... is an arthrit attempt at direct by calli k...
871	looking aristocratic , luminous yet careworn i...	1	look aristocrat , lumin yet careworn in jane h...

872 rows × 3 columns

Reading and preprocessing SST-2 dataset

```
from sklearn.feature_extraction.text import CountVectorizer
```

```
vectorizer = CountVectorizer()
train_X = vectorizer.fit_transform(train_df['preprocessed'])
display(train_X)
```

```
<67349x10106 sparse matrix of type '<class 'numpy.int64'>'
	with 535539 stored elements in Compressed Sparse Row format>
```

```
dev_X = vectorizer.transform(dev_df['preprocessed'])
display(dev_X)
```

```
<872x10106 sparse matrix of type '<class 'numpy.int64'>'
	with 12939 stored elements in Compressed Sparse Row format>
```

PyTorch Datasets and DataLoaders

PyTorch modules prefer to work with PyTorch **Datasets** and **DataLoaders**

A Pytorch Dataset

- Will extend torch.utils.data.Dataset
- Will primarily know how to yield one (x,y) item, given an index

A PyTorch DataLoader

- Will extend torch.utils.data.DataLoader
- Will know how to iterate over batches of items

For more info: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

PyTorch Datasets

```
class SST2Dataset(Dataset):
  def __init__(self,
               labels=None,
               sparse_count_matrix=None):

    self.y = torch.tensor(labels,dtype=torch.int64)

    self.X = sparse_count_matrix #Pytorch doesn't play especially well with
    # Sparse matrices, but we won't store the whole thing as a dense matrix

  def __len__(self):
    return self.y.shape[0]

  # The key method in a Dataset is __getitem__, which the DataLoader will
  # use to create batches
  def __getitem__(self, idx):
    rdict = {
      'y': self.y[idx],

      # Just densify individual rows of the sparse matrix as needed
      # A little awkward to convert it to a numpy array, but we want these
      # to be 1D vectors so that the DataLoader will stack them correctly
      'X': torch.Tensor(np.asarray(self.X[idx].todense())[0]),
    }
    return rdict
```

```
train_dataset = SST2Dataset(train_df['label'], train_X)
print(train_dataset[0])
print(train_dataset[0]['X'].shape)
```

```
{'y': tensor(0), 'X': tensor([0., 0., 0.,  ..., 0., 0., 0.])}
torch.Size([10106])
```

```
dev_dataset = SST2Dataset(dev_df['label'], dev_X)
print(dev_dataset[0])
```

```
{'y': tensor(1), 'X': tensor([0., 0., 0.,  ..., 0., 0., 0.])}
```

PyTorch DataLoaders

```python
batch_size = 16

# A DataLoader is a wrapper around a Dataset that makes it easy to iterate over
# batches of the dataset
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

# When we grab the first item in the iterator, it's a dictionary, but now each item
# is a batch of values from our dataset that has been stacked into a tensor, instead of a single value
first_train_batch = next(iter(train_dataloader))

print('First training batch:')
print(first_train_batch)

print('Batch item shapes:')
print({key:value.shape for key, value in first_train_batch.items()})
```

```
First training batch:
{'y': tensor([0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0]), 'X': tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]])}
Batch item shapes:
{'y': torch.Size([16]), 'X': torch.Size([16, 10106])}
```

PyTorch models

PyTorch models always extend torch.nn.Module

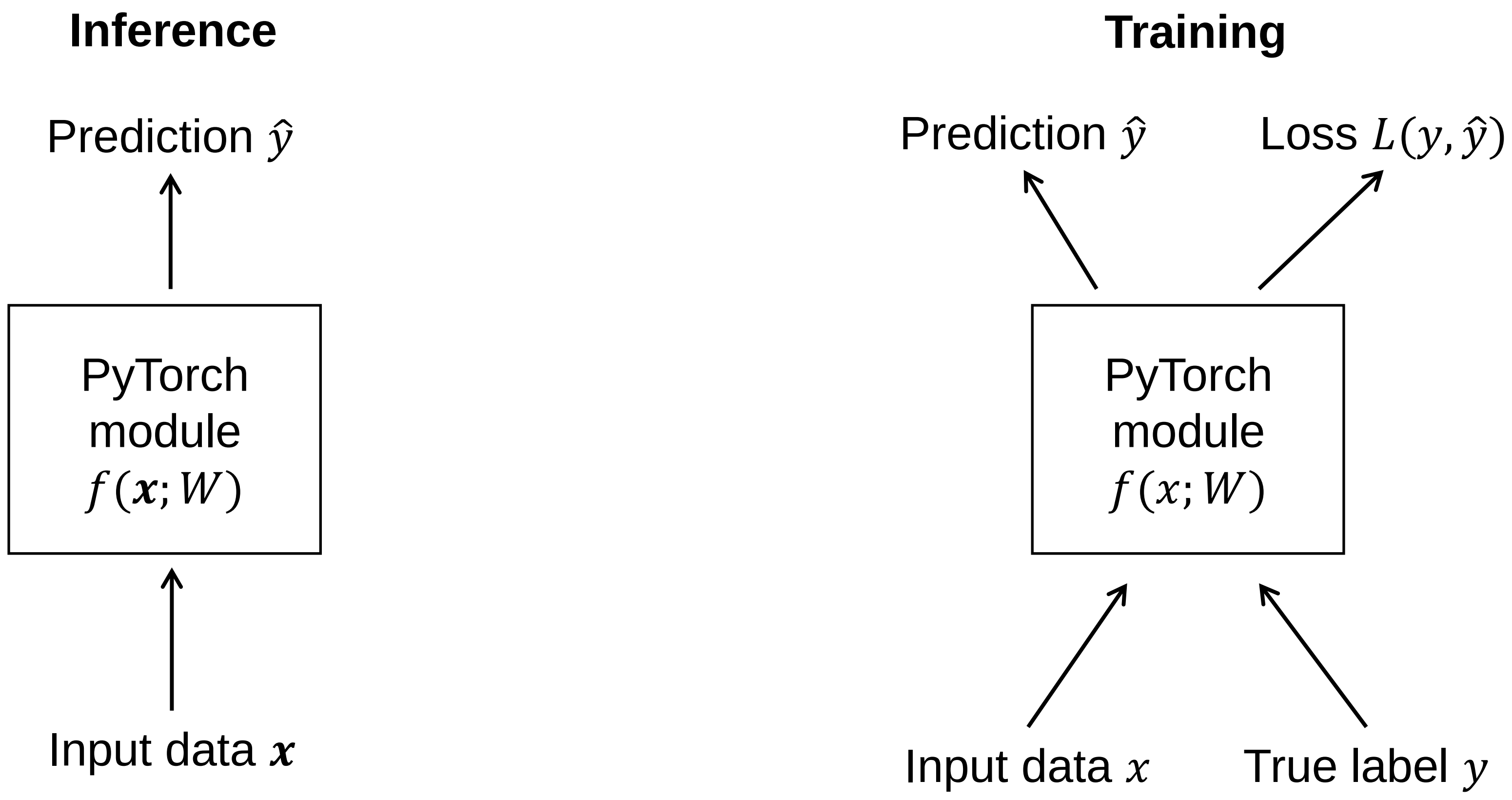
They always have:

- An `__init__()` method which defines the structure of the model
- A `forward()` method which takes in the input and spits out the model output

As long as the output of forward() is composed of differentiable tensor-on-tensor operations, then PyTorch can use **automatic differentiation** to figure out $\Delta_{parameters} output$, and then subsequently do gradient descent.

PyTorch models

A PyTorch model is essentially a wrapper around its forward() function, taking in an input tensor $\boldsymbol{x}$ and producing a prediction $\hat{y}$



Our model

```python
class BinaryLogisticRegression(torch.nn.Module):
  def __init__(self, vocab_size:int):
    super(BinaryLogisticRegression, self).__init__()
    self.Wb = torch.nn.Linear(vocab_size, 1, bias=True)
    self.activation_function = torch.nn.Sigmoid()

  def forward(self, X:torch.Tensor, y:torch.Tensor):
    py_logits = self.Wb(X)
    py_logits = py_logits.squeeze()
    py_probs = self.activation_function(py_logits)
    py = torch.round(py_probs).int()
    loss = torch.nn.functional.binary_cross_entropy(py_probs, y.float(), reduction ='mean')
    return {'py_logits':py_logits,
            'py_probs':py_probs,
            'py':py,
            'loss':loss}
```

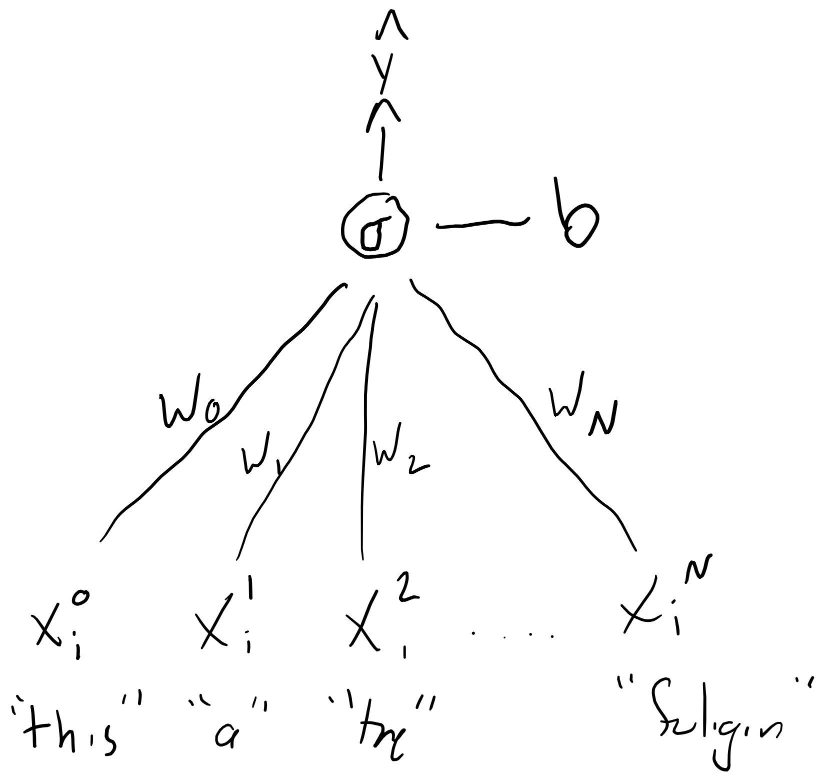
```python
# We can instantiate the model
vocab_size = train_X.shape[1] # We need to know this in order  to set up the model
our_model = BinaryLogisticRegression(vocab_size=vocab_size)
# Displaying the model will show its layers
display(our_model)
```

```
BinaryLogisticRegression(
  (Wb): Linear(in_features=10106, out_features=1, bias=True)
  (activation_function): Sigmoid()
)
```

Visualizing logistic regression

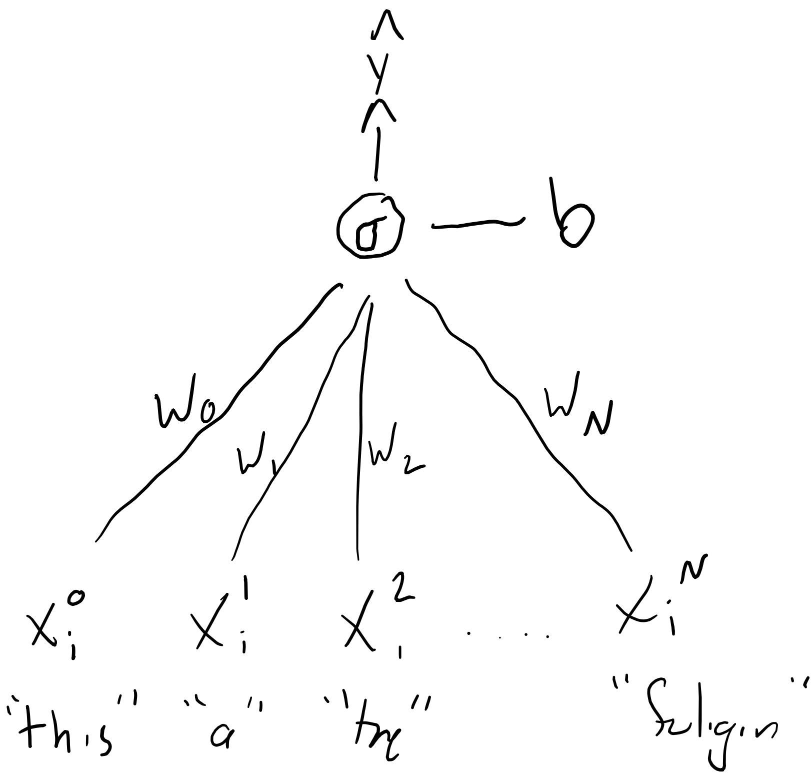
You can do the same thing for logistic regression by adding the σ function

$$\sigma\left(\begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^N \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^N \\ & & \cdots & & \\ x_M^0 & x_M^1 & x_M^2 & \cdots & x_M^N \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_N \end{bmatrix} + b\right) = \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \vdots \\ \hat{y}_M \end{bmatrix}$$



Our model

```python
class BinaryLogisticRegression(torch.nn.Module):
  def __init__(self, vocab_size:int):
    super(BinaryLogisticRegression, self).__init__()
    self.Wb = torch.nn.Linear(vocab_size, 1, bias=True)
    self.activation_function = torch.nn.Sigmoid()

  def forward(self, X:torch.Tensor, y:torch.Tensor):
    py_logits = self.Wb(X)
    py_logits = py_logits.squeeze()
    py_probs = self.activation_function(py_logits)
    py = torch.round(py_probs).int()
    loss = torch.nn.functional.binary_cross_entropy(py_probs, y.float(), reduction ='mean')
    return {'py_logits':py_logits,
            'py_probs':py_probs,
            'py':py,
            'loss':loss}
```




```python
# We can instantiate the model
vocab_size = train_X.shape[1] # We need to know this in order  to set up the model
our_model = BinaryLogisticRegression(vocab_size=vocab_size)
# Displaying the model will show its layers
display(our_model)
```

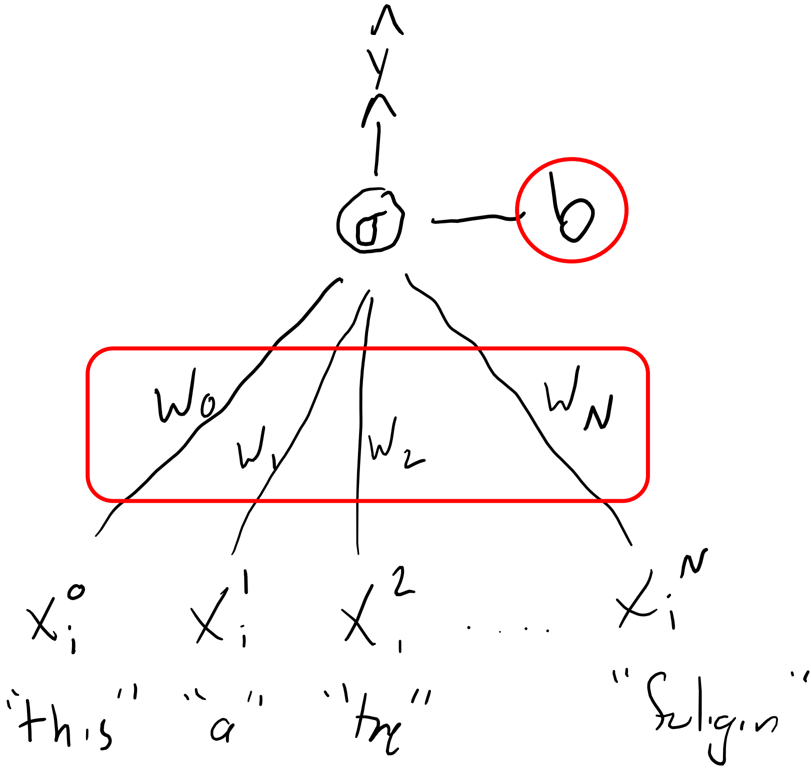
```
BinaryLogisticRegression(
  (Wb): Linear(in_features=10106, out_features=1, bias=True)
  (activation_function): Sigmoid()
)
```

Our model

```python
class BinaryLogisticRegression(torch.nn.Module):
    def __init__(self, vocab_size:int):
        super(BinaryLogisticRegression, self).__init__()
        self.Wb = torch.nn.Linear(vocab_size, 1, bias=True)
        self.activation_function = torch.nn.Sigmoid()

    def forward(self, X:torch.Tensor, y:torch.Tensor):
        py_logits = self.Wb(X)
        py_logits = py_logits.squeeze()
        py_probs = self.activation_function(py_logits)
        py = torch.round(py_probs).int()
        loss = torch.nn.functional.binary_cross_entropy(py_probs, y.float(), reduction ='mean')
        return {'py_logits':py_logits,
                'py_probs':py_probs,
                'py':py,
                'loss':loss}
```

```python
# We can instantiate the model
vocab_size = train_X.shape[1] # We need to know this in order  to set up the model
our_model = BinaryLogisticRegression(vocab_size=vocab_size)
# Displaying the model will show its layers
display(our_model)
```

```
BinaryLogisticRegression(
  (Wb): Linear(in_features=10106, out_features=1, bias=True)
  (activation_function): Sigmoid()
)
```

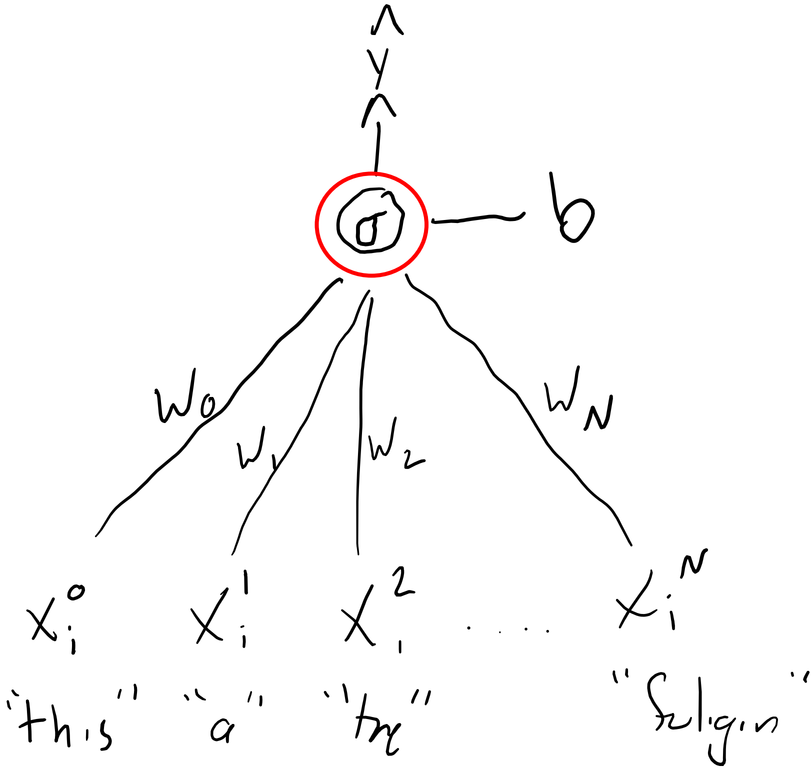


Our model

```python
class BinaryLogisticRegression(torch.nn.Module):
    def __init__(self, vocab_size:int):
        super(BinaryLogisticRegression, self).__init__()
        self.Wb = torch.nn.Linear(vocab_size, 1, bias=True)
        self.activation_function = torch.nn.Sigmoid()

    def forward(self, X:torch.Tensor, y:torch.Tensor):
        py_logits = self.Wb(X)
        py_logits = py_logits.squeeze()
        py_probs = self.activation_function(py_logits)
        py = torch.round(py_probs).int()
        loss = torch.nn.functional.binary_cross_entropy(py_probs, y.float(), reduction ='mean')
        return {'py_logits':py_logits,
                'py_probs':py_probs,
                'py':py,
                'loss':loss}
```

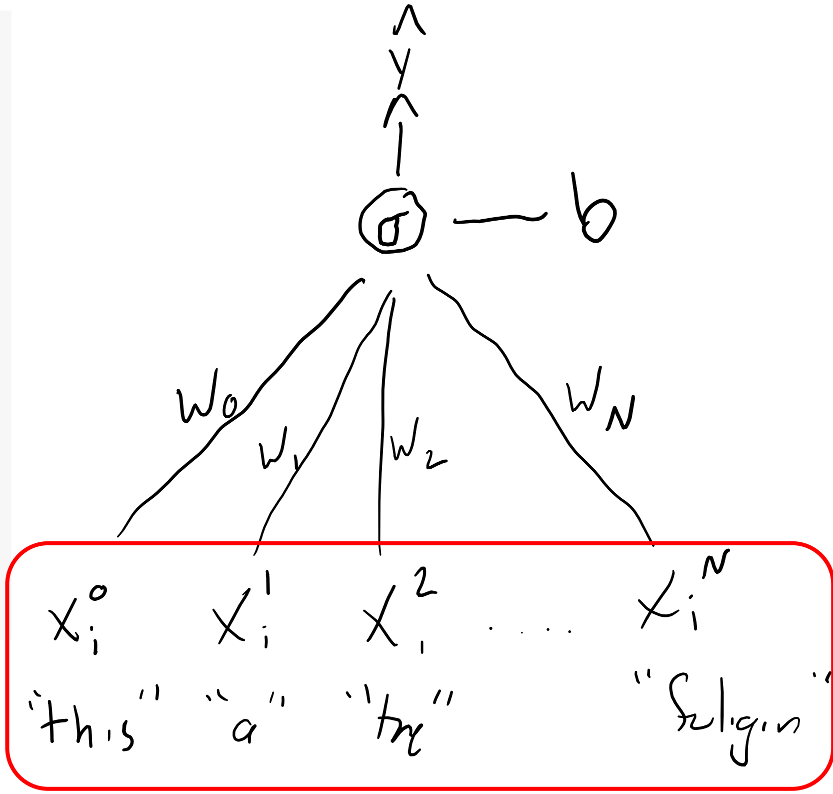
```python
# We can instantiate the model
vocab_size = train_X.shape[1] # We need to know this in order  to set up the model
our_model = BinaryLogisticRegression(vocab_size=vocab_size)
# Displaying the model will show its layers
display(our_model)
```

```
BinaryLogisticRegression(
  (Wb): Linear(in_features=10106, out_features=1, bias=True)
  (activation_function): Sigmoid()
)
```



Our model

```python
class BinaryLogisticRegression(torch.nn.Module):
  def __init__(self, vocab_size:int):
    super(BinaryLogisticRegression, self).__init__()
    self.Wb = torch.nn.Linear(vocab_size, 1, bias=True)
    self.activation_function = torch.nn.Sigmoid()

  def forward(self, X:torch.Tensor, y:torch.Tensor):
    py_logits = self.Wb(X)
    py_logits = py_logits.squeeze()
    py_probs = self.activation_function(py_logits)
    py = torch.round(py_probs).int()
    loss = torch.nn.functional.binary_cross_entropy(py_probs, y.float(), reduction ='mean')
    return {'py_logits':py_logits,
            'py_probs':py_probs,
            'py':py,
            'loss':loss}
```




```python
# We can instantiate the model
vocab_size = train_X.shape[1] # We need to know this in order  to set up the model
our_model = BinaryLogisticRegression(vocab_size=vocab_size)
# Displaying the model will show its layers
display(our_model)
```

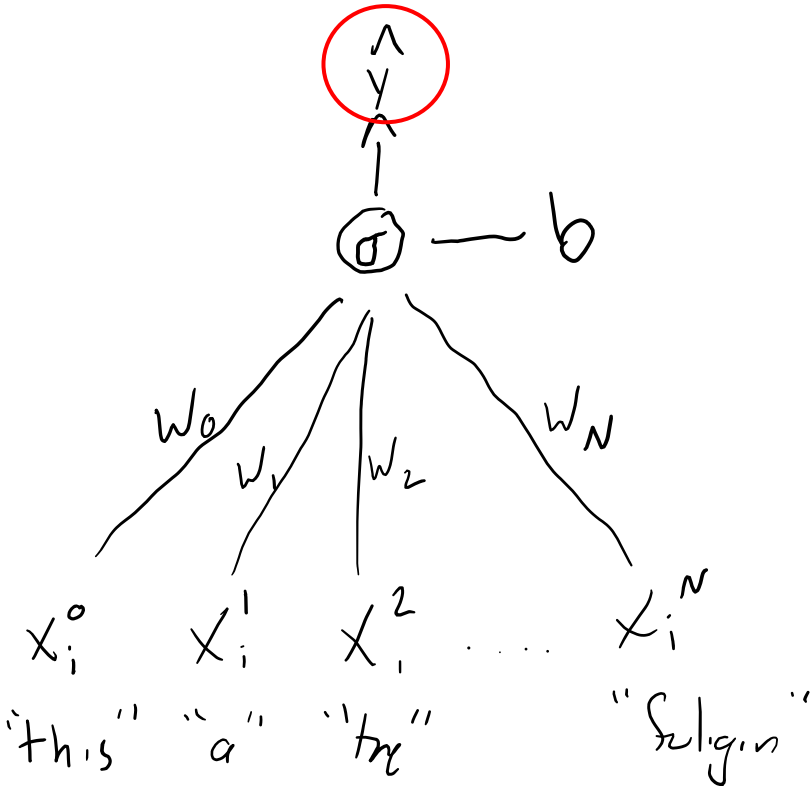
```
BinaryLogisticRegression(
  (Wb): Linear(in_features=10106, out_features=1, bias=True)
  (activation_function): Sigmoid()
)
```

Our model

```python
class BinaryLogisticRegression(torch.nn.Module):
  def __init__(self, vocab_size:int):
    super(BinaryLogisticRegression, self).__init__()
    self.Wb = torch.nn.Linear(vocab_size, 1, bias=True)
    self.activation_function = torch.nn.Sigmoid()

  def forward(self, X:torch.Tensor, y:torch.Tensor):
    py_logits = self.Wb(X)
    py_logits = py_logits.squeeze()
    py_probs = self.activation_function(py_logits)
    py = torch.round(py_probs).int()
    loss = torch.nn.functional.binary_cross_entropy(py_probs, y.float(), reduction ='mean')
    return {'py_logits':py_logits,
            'py_probs':py_probs,
            'py':py,
            'loss':loss}
```

```python
# We can instantiate the model
vocab_size = train_X.shape[1] # We need to know this in order  to set up the model
our_model = BinaryLogisticRegression(vocab_size=vocab_size)
# Displaying the model will show its layers
display(our_model)
```

```
BinaryLogisticRegression(
  (Wb): Linear(in_features=10106, out_features=1, bias=True)
  (activation_function): Sigmoid()
)
```



Our model

```python
from pprint import pprint
with torch.no_grad():
  first_train_output = our_model(**first_train_batch)

print('First training output:')
pprint(first_train_output)

print('Output item shapes:')
pprint({key:value.shape for key, value in first_train_output.items()})
```

```
First training output:
{'loss': tensor(0.6944),
 'py': tensor([1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1], dtype=torch.int32),
 'py_logits': tensor([ 0.0178, -0.0002, -0.0112,  0.0167, -0.0239,  0.0316, -0.0090,  0.0220,
         0.0039,  0.0079, -0.0467,  0.0296,  0.0232,  0.0018,  0.0024,  0.0018]),
 'py_probs': tensor([0.5045, 0.4999, 0.4972, 0.5042, 0.4940, 0.5079, 0.4977, 0.5055, 0.5010,
        0.5020, 0.4883, 0.5074, 0.5058, 0.5004, 0.5006, 0.5005])}
Output item shapes:
{'loss': torch.Size([]),
 'py': torch.Size([16]),
 'py_logits': torch.Size([16]),
 'py_probs': torch.Size([16])}
```

PyTorch training loop

Basic pseudocode:

For each epoch:

    For each training batch:

        Zero the accumulated grads

        Run model on training batch

        Calculate loss

        Perform gradient descent on step

    (optional)

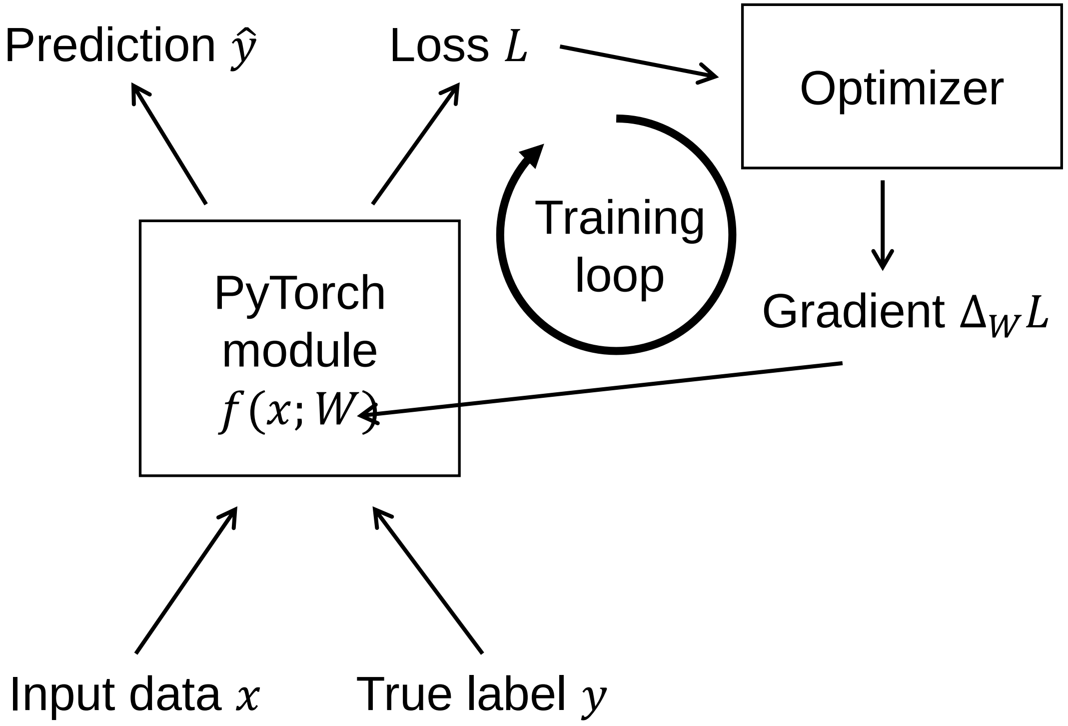
    For each validation batch:

        Run model on validation batch

    Report overall validation accuracy

PyTorch training loop

A PyTorch model is essentially a wrapper around its forward() function, taking in an input tensor x and producing a prediction y



Training loop

Preliminary stuff

```
learning_rate = 0.01

# We initialize an Adam optimizer with our chosen lr. There are other params
# we can set for Adam too, but I am ignoring them for now.
optimizer = torch.optim.Adam(our_model.parameters(), lr=learning_rate)
```

```
# An epoch is one complete pass over all the training batches
num_epochs = 2
```

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: 0.560
	Step 1000 mean training loss: 0.438
	Step 1500 mean training loss: 0.381
	Step 2000 mean training loss: 0.362
	Step 2500 mean training loss: 0.344
	Step 3000 mean training loss: 0.326
	Step 3500 mean training loss: 0.315
	Step 4000 mean training loss: 0.307
Epoch mean train loss: 0.374
Epoch train accuracy:0.849
Epoch dev accuracy:0.812

Epoch 1
	Step 500 mean training loss: 0.255
	Step 1000 mean training loss: 0.261
	Step 1500 mean training loss: 0.256
	Step 2000 mean training loss: 0.256
	Step 2500 mean training loss: 0.264
	Step 3000 mean training loss: 0.266
	Step 3500 mean training loss: 0.251
	Step 4000 mean training loss: 0.263
Epoch mean train loss: 0.258
Epoch train accuracy:0.902
Epoch dev accuracy:0.814
```

L1/L2 Regularization

Basic idea: discourage any one feature from having too much of an impact on the model output by punishing the sum (L1) or squared-sum (L2) of the model parameters

Standard way to discourage overfitting

Done automatically by most scikit-learn models

```
# We can see here that "kangaroo" totally sabotaged the prediction here
explain_binary_linear_model_prediction('the movie was wonderful and had a kangaroo in it.',
                                        lin_reg_model,
                                        vectorizer)
```

```
Prediction: -0.698151062283332
Word coefficients:
        Word: the - Coef: -0.005
        Word: movi - Coef: 0.002
        Word: wa - Coef: -0.079
        Word: wonder - Coef: 0.184
        Word: and - Coef: 0.024
        Word: had - Coef: -0.061
        Word: kangaroo - Coef: -1.353
        Word: in - Coef: -0.015
        Word: it - Coef: 0.003
Model intercept: 0.6021082139311205
```

Regularized model

```
class RegularizedBinaryLogisticRegression(torch.nn.Module):
  def __init__(self,
               vocab_size:int,
               l2_penalty_weight=.001):
    super(RegularizedBinaryLogisticRegression, self).__init__()
    self.Wb = torch.nn.Linear(vocab_size, 1, bias=True)
    self.activation_function = torch.nn.Sigmoid()
    self.l2_penalty_weight = l2_penalty_weight

  def forward(self, X:torch.Tensor, y:torch.Tensor):
    py_logits = self.Wb(X)
    py_logits = py_logits.squeeze()
    py_probs = self.activation_function(py_logits)
    py = torch.round(py_probs).int()
    py_loss = torch.nn.functional.binary_cross_entropy(py_probs, y.float(), reduction ='mean')

    l2_loss = self.l2_penalty_weight * torch.mean(self.Wb.weight**2)
    loss = py_loss+l2_loss

    return {'py_logits':py_logits,
            'py_probs':py_probs,
            'py':py,
            'loss':loss}
```

Regularized model

```python
# Then we can use the exact same training/evaluation loop on this new model
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):
    reg_optimizer.zero_grad()
    train_output = reg_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}')

      # The one thing I change here is to evaluate the dev loss more frequently so we can see how it's changing
      dev_pys = []
      dev_ys = []
      for dev_batch in dev_dataloader:
        with torch.no_grad():
          dev_output = reg_model(**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()
    reg_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}')
```

Regularized Model

Regularization didn't really help in this case. May be too simple model

We can observe we're doing a lot of extra work though…

```
Epoch 0
        Step 500 mean training loss: 0.562
        Dev accuracy:0.759
        Step 1000 mean training loss: 0.438
        Dev accuracy:0.811
        Step 1500 mean training loss: 0.381
        Dev accuracy:0.791
        Step 2000 mean training loss: 0.362
        Dev accuracy:0.805
        Step 2500 mean training loss: 0.339
        Dev accuracy:0.814
        Step 3000 mean training loss: 0.328
        Dev accuracy:0.819
        Step 3500 mean training loss: 0.313
        Dev accuracy:0.811
        Step 4000 mean training loss: 0.305
        Dev accuracy:0.822
Mean train loss: 0.375
Train accuracy:0.849
Dev accuracy:0.822
```

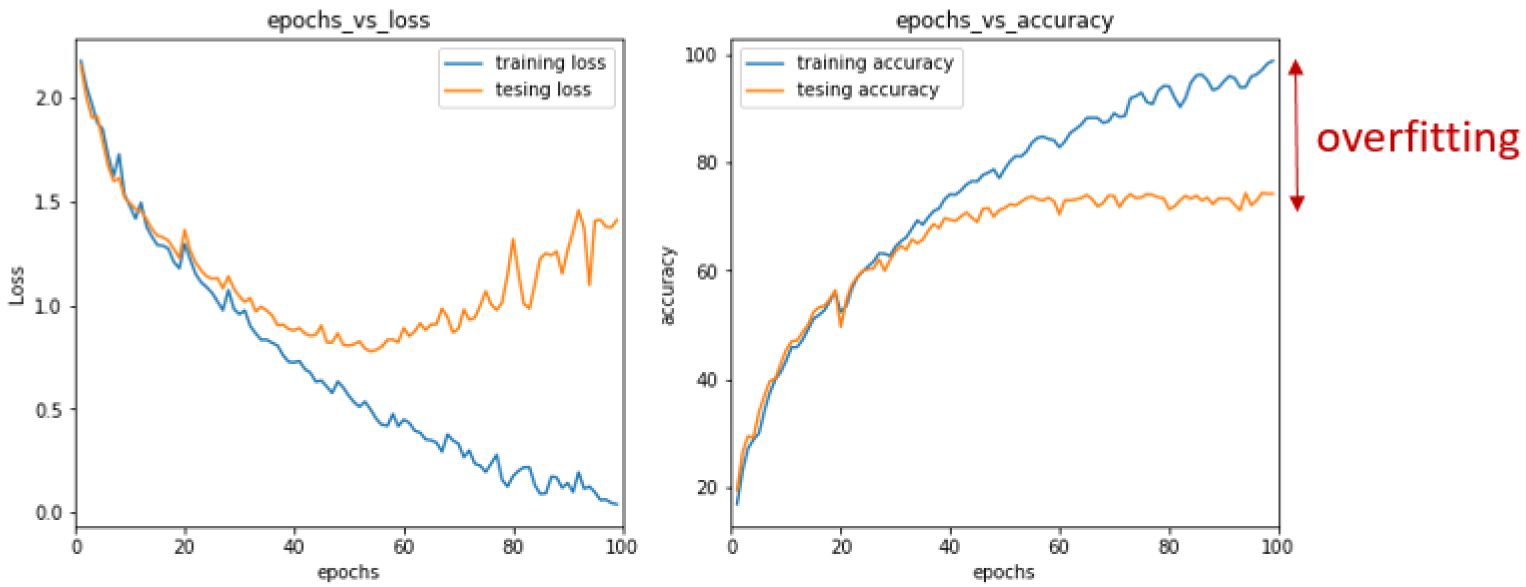
```
Epoch 1
        Step 500 mean training loss: 0.256
        Dev accuracy:0.820
        Step 1000 mean training loss: 0.262
        Dev accuracy:0.805
        Step 1500 mean training loss: 0.268
        Dev accuracy:0.823
        Step 2000 mean training loss: 0.255
        Dev accuracy:0.815
        Step 2500 mean training loss: 0.263
        Dev accuracy:0.826
        Step 3000 mean training loss: 0.258
        Dev accuracy:0.819
        Step 3500 mean training loss: 0.253
        Dev accuracy:0.815
        Step 4000 mean training loss: 0.257
        Dev accuracy:0.817
Mean train loss: 0.260
Train accuracy:0.903
Dev accuracy:0.817
```

Early stopping

Basic idea: Keep an eye on the development set performance (either loss or accuracy), and stop the training loop early when the improvement seems to level off

- Often save model checkpoints only on improvement, and then reload best checkpoint at the end of training

Another way to avoid overfitting



https://neptune.ai/blog/early-stopping-with-neptune

Early stopping

```
patience=3
best_dev_acc= 0.0
intervals_since_improvement=0
early_stop = False
```

...

```
    #Early stopping logic
    dev_acc = accuracy_score(np.concatenate(dev_ys), np.concatenate(dev_pys))
    print(f'\tDev accuracy:{dev_acc:.3f}')
    if dev_acc > best_dev_acc:
      best_dev_acc = dev_acc
      intervals_since_improvement =0
    else:
      intervals_since_improvement +=1

    if intervals_since_improvement > patience:
      print('Stopping early!')
      early_stop = True
      break
```

...

```
Epoch 0
	Step 500 mean training loss: 0.556
	Dev accuracy:0.782
	Step 1000 mean training loss: 0.438
	Dev accuracy:0.783
	Step 1500 mean training loss: 0.388
	Dev accuracy:0.803
	Step 2000 mean training loss: 0.361
	Dev accuracy:0.807
	Step 2500 mean training loss: 0.350
	Dev accuracy:0.802
	Step 3000 mean training loss: 0.321
	Dev accuracy:0.818
	Step 3500 mean training loss: 0.308
	Dev accuracy:0.827
	Step 4000 mean training loss: 0.302
	Dev accuracy:0.820
Mean train loss: 0.374
Train accuracy:0.850

Epoch 1
	Step 500 mean training loss: 0.251
	Dev accuracy:0.818
	Step 1000 mean training loss: 0.267
	Dev accuracy:0.817
	Step 1500 mean training loss: 0.268
	Dev accuracy:0.820
Stopping early!
```

Concluding thoughts

PyTorch: Machine learning Legos

Mini-batch gradient descent

- Batch size very important

Training loop

Avoid overfitting by:

- Regularization
- Early stopping