General Ingredients for Pytorch

1. The model (defined in an `nn.module` object)

2. The loss function

3. The training loop
General Ingredients for Pytorch

1. The model (defined in an `nn.module` object) maps $X$ to $y_{\text{pred}}$

2. The loss function evaluates $y_{\text{pred}}$ versus $y$

3. The training loop runs the model and loss in loop with gradient descent.
1. The model

maps $X$ (features) to $y_{pred}$ (prediction of $y$)

class LogReg(nn.Module):
    
def __init__(self, num_feats, learn_rate = 0.01, device = torch.device("cpu")):
        #the constructor; define any layer objects (e.g. Linear)
        super(LogReg, self).__init__()
        self.linear = nn.Linear(num_feats+1, 1) #add 1 to features for intercept

    def forward(self, X):
        #This is where the model itself is defined.
        #For binary logistic regression the model takes in $X$ and returns
        #a probability (a value between 0 and 1)

        newX = torch.cat((X, torch.ones(X.shape[0], 1)), 1) #add intercept

        return 1/(1 + torch.exp(-self.linear(newX))) #log func on the linear output
1. The model

maps $X$ (features) to $y_{pred}$ (prediction of $y$)

class LogReg(nn.Module):

    def __init__(self, num_feats, learn_rate = 0.01, device = torch.device("cpu")):
        # the constructor; define any layer objects (e.g. Linear)
        super(LogReg, self).__init__()
        self.linear = nn.Linear(num_feats+1, 1) # add 1 to features for intercept

    def forward(self, X):
        # This is where the model itself is defined.
        # For binary logistic regression the model takes in $X$ and returns
        # a probability (a value between 0 and 1)

        newX = torch.cat((X, torch.ones(X.shape[0], 1)), 1) # add intercept

        return 1/(1 + torch.exp(-self.linear(newX))) # log func on the linear output
"log loss" or "normalized log loss":

\[ J(\beta) = -\frac{1}{N} \sum_{i=1}^{N} y_i \log p(x_i) + (1 - y_i) \log (1 - p)(x_i) \]
2. The loss function *evaluates* $\texttt{ypred}$ *versus* $\texttt{y}$

```python
# e.g.
def normalizedLogLoss(ypred, ytrue):
    # Given:
    # ypred - a vector (torch 1-d tensor) of predictions from the model.
    # these are probabilities (values between 0 and 1)
    # ytrue - a vector (torch 1-d tensor) of the true labels
    # Output:
    # the logloss
    logloss = -1*torch.sum(ytrue*torch.log(ypred) + (1 - ytrue)*torch.log(1-ypred))
    N = ytrue.shape[0]
    normlogloss = (1/N)*logloss
    return normlogloss

# Alternative: return torch.nn.BCELoss(size_average=True)(ypred, ytrue)
```
"log loss" or "normalized log loss":

$$J(\beta) = -\frac{1}{N} \sum_{i=1}^{N} y_i \log p(x_i) + (1 - y_i) \log (1 - p)(x_i)$$
3. The training loop

runs the model and loss in loop with gradient descent.

#runs the training loop of pytorch model:
sgd = torch.optim.SGD(model.parameters(), lr=learning_rate) #gradient descent
loss_func = nn.CrossEntropyLoss() #includes log

#training loop:
for i in range(epochs):
  model.train() #tells pytorch we are training
  sgd.zero_grad() #sets the gradients to 0

  #forward pass:
  ypred = model(Xtrain)
  loss = loss_func(ypred, ytrain)

  #backward pass: runs gradient descent (or variant)
  loss.backward() #computes gradients
  sgd.step() #updates parameters

  if i % 20 == 0:
    print("  epoch: %d, loss: %.5f" %(i, loss.item()))
#training loop:
for i in range(epochs):
    model.train() # tells pytorch we are training
    sgd.zero_grad() # sets the gradients to 0

# forward pass:
ypred = model(Xtrain)
loss = loss_func(ypred, ytrain)

# backward pass: runs gradient descent (or variant)
loss.backward() # computes gradients
sgd.step() # updates parameters

if i % 20 == 0:
    print(" epoch: %d, loss: %.5f" %(i, loss.item()))
Training is done: how do I get predictions?

Easy!
Training is done: how do I get predictions?

Easy!

\[ \text{ypred} = \text{model}(X) \]
From binary logistic regression to multiclass softmax

Two updates

- Model (forward method)
- Loss function
Pytorch Specifics: Model

class LogReg(nn.Module):
    ...

    def forward(self, X):
        # This is where the model itself is defined.
        # For logistic regression the model takes in X and returns
        # the results of a decision function

        newX = torch.cat((X, torch.ones(X.shape[0], 1)), 1)  # add intercept

        return 1/(1 + torch.exp(-self.linear(newX)))
            # logistic function on the linear output
class MultiClassLogReg(nn.Module):
    def __init__(self, num_feats, num_classes, learn_rate = 0.01, device = torch.device("cpu") ):
        #the constructor; define any layer objects (e.g. Linear)
        super(LogReg, self).__init__()
        self.linear = nn.Linear(num_feats+1, num_classes)

    def forward(self, X):
        #This is where the model itself is defined.
        #For logistic regression the model takes in X and returns
        #the results of a decision function

        newX = torch.cat((X, torch.ones(X.shape[0], 1)), 1) #add intercept

        #return 1/(1 + torch.exp(-self.linear(newX)))
        #logistic function on the linear output

        return self.linear(newX) #only use linear if using cross-entropy loss
Pytorch Specifics: loss

# runs the training loop of pytorch model:
sgd = torch.optim.SGD(model.parameters(), lr=learning_rate)
loss_func = nn.CrossEntropyLoss() # includes log

# training loop:
for i in range(epochs):
    model.train()
    sgd.zero_grad()
    # forward pass:
    ypred = model(X)
    loss = loss_func(ypred, y)
    # backward: /(applies gradient descent)
    loss.backward()
    sgd.step()

    if i % 20 == 0:
        print("  epoch: %d, loss: %.5f"%(i, loss.item()))
Two equivalent options for multi-class:

**option 1 (what the previous slides covered)**

```python
# in model/forward:
    return self.linear(newX)  # only use linear if using cross-entropy loss

# in loss/train:
    loss_func = nn.CrossEntropyLoss()  # includes log softmax
        # alternative: nn.NLLLoss()  # negative log likelihood loss
```

**option 2**

```python
# in model/forward:
    return nn.log_softmax(self.linear(newX))  # log softmax is multiclass

# in loss/train:
    loss_func = nn.NLLLoss()  # negative log likelihood loss
```