# Deep Learning

## Cross Entropy Loss in PyTorch

Question: for Cross Entropy Loss calculation in PyTorch, why the input doesn't need to be probabilities when the CE calculation requires probabilities?

Answer: the PyTorch calculation is a little bit different, it has SoftMax calculation built in in the cross-entropy loss calculation

Cross entropy loss (summation is over all classes)

$$H(p,q) = -\sum p(x) * \log\big(q(x)\big)$$

Where $q(x)$ is the predicted probability after SoftMax calculation

How PyTorch calculates cross entropy loss (summations are overall classes)

$$l_n = -\sum w_c * \log\frac{e^{x_{n,c}}}{\sum e^{x_{n,i}}} * y_{n,c}$$

## Precision and Recall

Question: what is the difference between precision and recall and accuracy?

Answer:

- Accuracy is the number of correct classifications out of all samples
- Recall is the number of correct positive cases identified out of all positive cases

$$recall = \frac{true\ positives}{true\ positives\ + false\ negatives} = \frac{terrorists\ correctly\ identified}{terrorists\ correctly\ identified + terrorists\ incorrectly\ labeled\ as\ not\ terrorists}$$

- Precision is the number of correct positive cases identified out of positive cases identified by the model

$$precision = \frac{true\ positives}{true\ positives + false\ positives} = \frac{terrorists\ correctly\ identified}{terrorists\ correctly\ identified + individuals\ incorrectly\ labeled\ as\ terroists}$$

## *Torch.nn.Embedding*

Syntax: $torch.nn.embedding(vocab\_size, embed\_dim)$

- Each value in the input_text must be < vocabulary size
- The layer takes each value and convert the value from one number to vector embedding of size embed_dim
- The layer is basically a lookup table, the value represent the index to retrieve the vector embedding

Assuming vocabulary size of 4 and embedding dimension of 3

- Given the weight matrix

```
tensor([[ 0.3839,  0.3059, -0.2729],
        [ 0.1917, -0.0568, -0.4838],
        [-0.0663,  0.2103,  0.4577],
        [ 0.0898,  0.1073,  0.0337]])
```

- The input text of [3,2,1] will be embedded into

```
tensor([[ 0.0898,  0.1073,  0.0337],
        [-0.0663,  0.2103,  0.4577],
        [ 0.1917, -0.0568, -0.4838]])
```

- This is equivalent to one hot encode the input and feed it to a linear layer with the transpose of the given weight matrix

```
input_text = torch.tensor([[0,0,0,1],
                           [0,0,1,0],
                           [0,1,0,0]],dtype=torch.float32)
output = w_linear(input_text)
output
```

✓ 0.0s

```
tensor([[ 0.0898,  0.1073,  0.0337],
        [-0.0663,  0.2103,  0.4577],
        [ 0.1917, -0.0568, -0.4838]], grad_fn=<MmBackward0>)
```

## *Param Initialization Part 1: Expectation of Initial Loss for classification*

- Assume n class, and parameters are initialized to predict roughly equal probability for each class, the cross entropy loss would be

$$-\ln(1/n)$$

## *Param Initialization Part 2: Effect of highly skewed initial prediction*

- When initial model output is highly skewed (not evenly distributed across all classes), the model will take a lot of unnecessary training to first make the distribution even, then train toward the correct class.
    - The outcome is so that the output logit (model output before the softmax layer or sigmoid layer) is small (closer to zero)
    - Hence, multiple the initial bias by 0 (initialize all bias to 0), multiple the initial W by 0.1 or 0.01
    - Why not set W to 0?

## *Param Initialization Part 3: consequence of highly activate tanh activation*

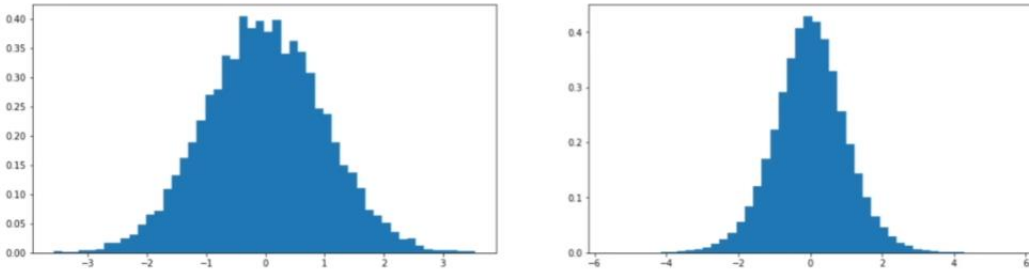
- When tanh is high activated, that implies that there are many extreme values got squashed into negative one and one, in this case, those values are highly inelastic – tweaking the parameters would not change the outcome after the activation, those neurons are close to be "dead"
    - This is like a permanent brain damage: the neuron will never activate because it is in the extreme of the manifold
    - In this case, we want to make sure that the distribution of the data feed into the activation (at least initially) is small. This means the weights should be very small and close to 0

## *Param Initialization Part 4: how to initialize*

- How to preserve the input standard deviation with matrix multiplication (what number should we multiple W with?)

```
In [536]: x = torch.randn(1000, 10)
          w = torch.randn(10, 200) / 10**0.5
          y = x @ w
          print(x.mean(), x.std())
          print(y.mean(), y.std())
          plt.figure(figsize=(20, 5))
          plt.subplot(121)
          plt.hist(x.view(-1).tolist(), 50, density=True);
          plt.subplot(122)
          plt.hist(y.view(-1).tolist(), 50, density=True);

          tensor(-0.0165) tensor(1.0018)
          tensor(-0.0051) tensor(1.0069)
```



- $1/\sqrt{n}$ – n is the number of features / inputs

- Paper:
  - Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (2015)
- PyTorch:
  - Kaiming_normal

## *Batch Normalization*

- It is unnatural because the model is supposed to process a single input, we train them in batches because of optimization. Batch normalization normalizes the layer output for each batch, then use parameters to make shifts and scale to the distribution.
  - The layer output of a sample will be a little different when it is trained in a different batch
  - This acts like a regularization (a little like data augmentation)

- For inference time
  - Idea 1: calibrate the batch norm at the end of training - Calculate the mean and std over the entire training set to be used at inference time
  - **Idea 2**: keep track of the running stats – initialize running mean at 0 and running std at 1 with size equals to (1, hidden dim)

$$with\ torch.no\_grad$$

$$mean_{running} = 0.999 \times mean_{running} + 0.001 \times mean_{current\ batch}$$

    - Std calculation is the same
    - Receives a small update every time

        - The 0.001 is the momentum, roughly speaking, if you have a large batch size, we can use higher momentum, because the batch mean and std will be roughly the same for different batches if the batch size is large

- The layer before the batch norm should not have bias term
    - Since we are standardizing the data with batch normalization, adding the bias term is a waste, for example, the bias shifts the distribution rightward, then the normalization step will shift it leftward if the distribution is centered on a positive value

## *Implementation*

### *Data type for PyTorch code*

- For binary cross entropy loss with logit
    - The target labels have to be in float as opposed to int