Classification

Consider a multi-class classification task:

$\{(x_i, y_i)\}_{i=1}^{\ell}$, $y_i \in \{1, \dots, C\}$, C - number of classes

Ideally, we would like to minimize the error rate:

$$\frac{1}{\ell} \sum_{i=1}^{\ell} [a_w(x_i) \neq y_i] \to \min_w$$

indicator function, =1 if condition holds, =0 otherwise

However, the derivative of indicator either doesn't exist or equals zero (piecewise constant function)

Thus, we need a smooth loss function to train our classifier

Let the network predict **class probabilities**

Let $z = (z_1, \dots, z_C) \in \mathbb{R}^C$ be a **logit vector** (output of the last linear layer without activation, size equal to number of classes C)

Define probabilities: $\forall k \in \{1, \dots, C\}$

$$p(y=k) = p_k := \text{Softmax}(z)_k = \frac{\exp(z_k)}{\sum_{j=1}^{C} \exp(z_j)}$$

z - logit vector

p - probabilites vector

$p_k \geq 0$

$\sum_{k=1}^{C} p_k = 1$

Train the network with maximum likelihood method:

Loss for a single object:

$$-\sum_{k=1}^{C} [k=y] \log p_k = -\log p_y \to \min$$

Loss over whole train set:

$\{(x_i, y_i)\}_{i=1}^{\ell}$ $(p_1^{(i)}, \dots, p_C^{(i)})$ – probabilities predicted for x_i by the network

$$\mathcal{L} = -\frac{1}{\ell} \sum_{i=1}^{\ell} \sum_{k=1}^{C} [y_{(i)}=k] \log p_k^{(i)} \to \min$$

Negative log-likelihood (NLLLoss)

In practice it is more numerically stable to compute LogSoftmax as a single operation rather apply log to Softmax result:

$$\text{LogSoftmax}(z)_k = \log \frac{\exp(z_k)}{\sum_{j=1}^{C} \exp(z_j)} =$$

$$= \log \exp(z_k) - \log \left(\sum_{j=1}^{C} \exp(z_j)\right) =$$

$$= z_k - \underbrace{\log \left(\sum_{j=1}^{C} \exp(z_j)\right)}_{\text{Log Sum Exp}}$$

To compute LogSumExp in a numerically stable way, we need to get rid of large exponents

$$\log\left(\sum_{j=1}^{C}\exp(z_j)\right) = \log\left(\sum_{j=1}^{C}\exp(z_j - \max_m z_m + \max_m z_m)\right) =$$

$$= \log\left(\exp(\max_m z_m)\cdot\sum_{j=1}^{C}(z_j - \max_m z_m)\right) =$$

$$= \log\left(\exp(\max_m z_m)\right) + \log\left(\sum_{j=1}^{C}\exp(z_j - \max_m z_m)\right) =$$

$$= \max_m z_m + \log\left(\sum_{j=1}^{C}\exp(z_j - \max_m z_m)\right)$$

$$\Rightarrow \text{Log Softmax}(z)_k = z_k - \max_m z_m + \log\left(\sum_{j=1}^{C}\exp(z_k - \max_m z_m)\right)$$

Similarly, if we want to compute Softmax itself, we can use the following formula:

$$\text{Softmax}(z)_k = \frac{\exp(z_k - \max_m z_m)}{\sum_{j=1}^{C}\exp(z_j - \max_m z_m)}$$

NLL Loss is a special case of Cross-entropy, which is a similarity between 2 arbitrary probability distributions $\{q_k\}_{k=1}^{C}$, $\{p_k\}_{k=1}^{C}$

$$CE(p, q) = -\sum_{k=1}^{C} q_k \log p_k$$

In our case $q_k = [k = y]$ – degenerate distribution

In PyTorch:

- NLLLoss – takes LogSoftmax output and targets
- CrossEntropyLoss – takes logits and targets

Dropout

Main idea: zero out random neurons to prevent the network from overfitting (regularization technique)

Let $X = (X_1, \dots, X_i, \dots, X_B)^T \in \mathbb{R}^{B \times N}$ be a batch of latent representations (i.e. an intermediate linear layer output)

B - batch size

N - number of features

(We pass the whole batch through the network, not the objects one-by-one)

Sample a binary mask $m \in \{0, 1\}^{B \times N}$

$m_{ij} \sim \text{Bernoulli}(1-p)$ p - dropout rate

typically $0.1 \le p \le 0.5$

Train mode:

1. Sample new mask m
2. $y = m \odot x \cdot \frac{1}{1-p}$

↖ element-wise product

- Normalization by $\frac{1}{1-p}$ is required to preserve the expectation of neurons:

$$\mathbb{E}[y] = \mathbb{E}\left[m \odot x \cdot \frac{1}{1-p}\right] = \frac{1}{1-p} \cdot x \odot \mathbb{E}[m] =$$

$$= \frac{1}{1-p} \cdot x \cdot (1-p) = x$$

- The mask needs to be stored for the backward pass.

Eval mode:

As we normalized the output during training, dropout is an identical transform during testing:

1. $y = x$

- Dropout makes training slower but may improve test quality

Batch normalization

Main idea: stabilize training by forcing zero mean and unit variance of features

$X \in \mathbb{R}^{B \times N} = (x_1, \ldots, x_B)^T$

Train mode:

1. Compute batch statistics and normalize x:

$$\mu = \frac{1}{B}\sum_{i=1}^{B} x_i \qquad \sigma^2 = \frac{1}{B}\sum_{i=1}^{B} \underbrace{(x_i - \mu)^2}_{\text{element wise square}} \qquad \mu, \sigma^2 \in \mathbb{R}^N$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

small correction for numeric stability

- Note that all operations are element wise

- All operations are differentiable, so we need to compute $\frac{dl}{d\hat{x}_i}, \frac{dl}{d\mu}, \frac{dl}{d\sigma^2}, \frac{dl}{dx_i}$ during backward pass

2. Apply trainable elementwise affine transform:

$$y_i = \hat{x}_i \odot w + b \; ; \; w, b \in \mathbb{R}^N$$

these parameters are trained with gradient descent

- We give the model freedom to set the desired mean and variance for x_i.

3. During test we typically do not have access to batch of data (we must be able to predict for a single test object)

Thus, we will accumulate statistics from the training batches:

$$\text{running\_mean} := \text{running\_mean} \cdot (1-m) + \mu \cdot m$$

$$\text{running\_var} := \text{running\_var} \cdot (1-m) + \sigma^2 \cdot m \cdot \frac{B}{B-1}$$

make variance estimate unbiased

m – momentum parameter (typically $m=0.1$ to ensure slow updates)

At the beginning of training, running_mean is initialized with zeros and running_var with ones.

These vectors are not counted as trainable parameters

Eval mode:

1. Normalize using accumulated statistics:

$$\hat{x}_i = \frac{x_i - \text{running\_mean}}{\sqrt{\text{running\_var} + \varepsilon}}$$

2. Apply affine transform:

$$y_i = \hat{x}_i \odot w + b$$

- Note: the network can't learn proper normalization vectors itself, as it is trained with gradient descent. By normalizing the features manually, we ensure that gradients are adequate (not too large or too small)