UACM
Universidad Autónoma
de la Ciudad de México
NADA HUMANO ME ES AJENO

Redes Neuronales

Sabino Miranda

Funciones de pérdida

Funciones de pérdida

- Una función de pérdida o función de costo o función de error mide la diferencia entre las predicciones del modelo y los valores esperados (reales).
- El objetivo del entrenamiento es minimizar esta función, ajustando los parámetros del modelo para mejorar la precisión.
- Error Cuadrático Medio (MSE)
- Entropía Cruzada (Cross-Entropy)
- Kullback-Leibler Divergence (KL Divergence)

Error Cuadrático Medio (MSE) (1)

- El error cuadrático medio (*mean squared error, MSE*) mide la diferencia promedio al cuadrado entre las predicciones del modelo y los valores reales.
- Cuanto menor sea el valor de la MSE, mejor será el desempeño del modelo, ya que indica que las predicciones están más cerca de los valores reales.
- Para N ejemplos, el error cuadrático medio se calcula como:

$$\text{MSE} = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

donde:

  - N es el número total de ejemplos

Error Cuadrático Medio (MSE) (2)

- y_i es el valor real (esperado) para el i-ésimo ejemplo
- $\hat{y}_i$ es la predicción del modelo para el i-ésimo ejemplo.

- En PyTorch, para el error cuadrático medio se utiliza la función de pérdida `MSELoss`.

```
criterion = nn.MSELoss()
```

Entropía Cruzada (1)

- La Entropía Cruzada (o Cross-Entropy) es una medida de la diferencia entre dos distribuciones de probabilidad.
- Es muy utilizada como función de pérdida en problemas de clasificación, tanto en clasificación binaria como multiclase.
- Mide cuán bien el modelo predice las etiquetas correctas, penalizando las predicciones incorrectas de manera más severa cuando están muy alejadas del objetivo.

Entropía Cruzada (2)

- Para clasificación binaria, donde las etiquetas son 0 o 1, la entropía cruzada se define como:

$$H(p, q) = -\left(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right)$$

donde:

- y es la etiqueta real (0 o 1),
- $\hat{y}$ es la probabilidad predicha
- log es el logaritmo natural.

Entropía Cruzada (3)

- En problemas de clasificación multiclase, donde hay más de dos clases, la entropía cruzada se define como:

$$H(p, q) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

donde:

- C es el número de clases,
- $\hat{y}_i$ es la probabilidad predicha para la clase i.
- y_i es la etiqueta real para la clase i (en codificación one-hot),

Entropía Cruzada (4)

- En PyTorch, para la clasificación binaria se utiliza la función de pérdida `BCELoss`, que es la entropía cruzada binaria.

```
criterion = nn.BCELoss()
```

- Para clasificación multiclase, se utiliza la función de pérdida `CrossEntropyLoss`, que calcula la entropía cruzada multiclase.

```
criterion = nn.CrossEntropyLoss()
```

Entropía Cruzada (5)

- Propiedades:
  - La entropía cruzada es mínima cuando las probabilidades predichas $\hat{y}_i$ coinciden con las etiquetas reales y_i.
  - Valor máximo: La entropía cruzada aumenta cuando las probabilidades predichas son muy diferentes de las verdaderas etiquetas.
  - Penaliza fuertemente las predicciones incorrectas cuando las probabilidades son muy diferentes de las etiquetas reales.

Entropía Cruzada (6)

- En Pytorch, la función `CrossEntropyLoss` espera como entrada las activaciones de la última capa de la red, no las probabilidades normalizadas (softmax). Internamente, realiza el cálculo de la función log-softmax, aplica la entropía cruzada en una sola operación.

$$\mathcal{L}(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log\left(\frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}\right) \tag{1}$$

donde:

  - z_i son los **logits**, es decir, las salidas sin activar de la red.
  - $\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$ corresponde a la salida *softmax*, interpretada como la probabilidad estimada de la clase i.

Entropía Cruzada (7)

- y_i representa la etiqueta verdadera en formato *one-hot* (1 si la clase es correcta, 0 en otro caso).

Forma simplificada: Cuando sólo se considera la clase verdadera c, la pérdida se reduce a:

$$\mathcal{L}(y, \hat{y}) = -\log\left(\frac{e^{z_c}}{\sum_{j=1}^{C} e^{z_j}}\right) \tag{2}$$

Forma para un lote (*batch*) de N ejemplos:

$$\mathcal{L}_{\text{batch}} = -\frac{1}{N}\sum_{n=1}^{N}\log\left(\frac{e^{z_{c_n}^{(n)}}}{\sum_{j=1}^{C} e^{z_j^{(n)}}}\right) \tag{3}$$

Códificación One-hot (1)

- La codificación one-hot es una técnica de representación que se utiliza en el procesamiento de datos, comúnmente en problemas de clasificación.
- Transformar datos categóricos en una forma numérica que un modelo pueda interpretar.
- Consiste en representar cada categoría única de una variable como un vector binario, en el cual solo una posición del vector es 1 y las demás son 0.
- Supongamos que tenemos una variable categórica con tres categorías: *positivo*, *negativo* y *neutro*.

Códificación One-hot (2)

- Se asigna a cada categoría un vector binario con la misma longitud que el número de categorías.
  Por ejemplo:

Categoría	**Codificación One-Hot**
positivo	[1, 0, 0]
negativo	[0, 1, 0]
neutro	[0, 0, 1]

Tabla: Ejemplo de codificación one-hot

- La codificación *one-hot* tiene ventajas:

Códificación One-hot (3)

1. **Evita el sesgo de magnitud**: A diferencia de asignar números enteros (por ejemplo, positivo=1, negativo=2, neutro=3) *one-hot* asegura que no se introduzca un orden o una magnitud entre las categorías, todas se representan de manera equivalente.
2. **Facilita el cálculo en redes neuronales**: En modelos de aprendizaje automático, tener representaciones numéricas sin un orden subyacente permite que el modelo aprenda relaciones sin inferir alguna jerarquía numérica no deseada.

- Si el problema requiere un orden o jerarquía no usarlo. Por ejemplo, intensidad de lo positivo: 1, 2, 3, 4, 5.

Códificación One-hot (4)

- Por ejemplo en PyTorch:

```
import torch

# Supongamos que tenemos tres etiquetas para codificar: 0,
↪  1 y 2.
labels = torch.tensor([0, 1, 2])

# Convertir etiquetas a codificación one-hot para tres
↪  clases
one_hot_labels = nn.functional.one_hot(labels,
↪  num_classes=3)
print(one_hot_labels)
----
tensor([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

Divergencia de Kullback-Leibler (1)

- La **divergencia de Kullback-Leibler** (KL) es una medida de la diferencia entre dos distribuciones de probabilidad.
- Es una medida de divergencia dirigida en la que se mide la diferencia de una distribución de probabilidad P con respecto a una distribución de probabilidad de referencia Q.
- La divergencia KL para distribuciones discretas se define como:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_x P(x) \log\left(\frac{P(x)}{Q(x)}\right) = -\sum_x P(x) \log\left(\frac{Q(x)}{P(x)}\right)$$

Divergencia de Kullback-Leibler (2)

y para distribuciones continuas como:

$$D_{\mathrm{KL}}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log \left(\frac{p(x)}{q(x)} \right) dx$$

donde:

- $P(x)$ es la distribución verdadera (o real),
- $Q(x)$ es la distribución aproximada (o modelo),
- la suma o integral recorre todos los posibles valores de x.

- Propiedades:
  - La divergencia KL no es simétrica,
    $D_{\mathrm{KL}}(P \parallel Q) \neq D_{\mathrm{KL}}(Q \parallel P)$
  - $D_{\mathrm{KL}}(P \parallel Q) \geq 0$
  - Mide cuánta información adicional se necesita para aproximar la distribución P usando Q.

Divergencia de Kullback-Leibler (3)

- La divergencia de Kullback-Leibler (KL) se utiliza principalmente en tareas de regularización, aprendizaje no supervisado y modelos generativos.
- En clasificación, particularmente cuando se utilizan probabilidades de salida (como en el caso de la función de activación softmax), se puede utilizar la divergencia KL como función de pérdida en lugar de la entropía cruzada.

Divergencia de Kullback-Leibler (4)

- Por ejemplo en PyTorch:

```
import torch
import torch.nn as nn

logits = torch.randn(2, 3, requires_grad=True)  # 2
↪  ejemplos y 3 clases
softmax = nn.Softmax(dim=1)
log_probs = softmax(logits)

# Crear un tensor de probabilidades objetivo
target_probs = torch.tensor([[0.1, 0.2, 0.3],
                             [0.0, 0.0, 0.5]])

# Instanciar la función de pérdida KLDivLoss
kl_loss = nn.KLDivLoss(reduction="batchmean")

# Calcular la pérdida de Kullback-Leibler
loss = kl_loss(log_probs, target_probs)
print("Pérdida de Kullback-Leibler:", loss.item())
loss.backward()
print(logits.grad)
```

Clasificación de textos

- Construir una red neuronal como clasificador de textos con Pytorch
- Usar funciones de activación PReLU
- Usar codificación *one-hot*
- Usar funciones de pérdida CrossEntropy

Ejemplo: Construir una red neuronal como clasificador de textos con Pytorch

Consultar el notebook: 13_*PyTorch_MLP_Agresividad.ipynb*

Referencias (1)

1. Deep Learning. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. MIT Press, 2016. http://www.deeplearningbook.org
2. Dive into Deep Learning. Aston Zhang, Zachary C. Lipton, Mu li, and Alexander J. Smola. Cambridge University Press, 2023. https://d2l.ai
3. Neural Networks and Deep Learning A Textbook (2nd Edition). Charu C. Aggarwal. Springer, 2023. https://doi.org/10.1007/978-3-031-29642-0
4. Deep Learning: Foundations and Concepts. Christopher M. Bishop and Hugh Bishop. Springer, 2024. https://doi.org/10.1007/978-3-031-45468-4

Referencias (2)

5. PyTorch documentation.
   `https://pytorch.org`
6. Numpy documentation.
   `https://numpy.org`
7. Python documentation.
   `https://www.python.org`