16. travnja 2021.

Zavod za elektroniku, mikroelektroniku,
računalne i inteligentne sustave

Duboko učenje

provjera znanja 2. laboratorijske vježbe

1. Predložite izvedbe sučelja `Layer` za sljedeće slojeve:

   (a) Transformacija $F(\mathbf{x}) = \mathbf{A}(\mathbf{x}\odot\mathbf{x})+b\mathbf{x}+\mathbf{c}$, gdje su $\mathbf{A}$ i $\mathbf{c}$ parametri sloja, a b hiperparametar.

   (b) Aktivacijska funkcija Swish $g(x,\beta) = x \cdot \sigma(\beta x)$, gdje $\sigma(\cdot)$ predstavlja sigmoidu, a β hiperparametar.

   Vaše izvedbe utemeljite na biblioteci Numpy. Pretpostavite da na ulaze vaših slojeva dolaze minigrupe 1D podataka u obliku matrica dimenzija $N \times D$.

2. Nadopunite sljedeći kod tako da izvedete sloj sažimanja prosječnom vrijednošću u oknu veličine $k \times k$ s korakom k pomoću konvolucijskog sloja `nn.Conv2D` iz biblioteke Pytorch. Pretpostavite da na ulaz sloja dolaze minigrupe slika dimenzija N×C×H×W.

```
class ConvAvgPool2d(nn.Module):
  def __init__(self, channels, K):
    super(ConvAvgPool2d, self).__init__()
    self.K = K
    self.conv = nn.Conv2d(_, _, kernel_size=_, stride=_, padding=_, bias=_)
    self.conv.weight = self.init_weight()

  def init_weight(self):
    '''Returns weight tensor of shape (c_out, c_in, K, K)'''

  def parameters(recurse=True):
    '''Returns layer's parameters'''

  def forward(self, x):
    '''Performs forward pass'''
```

3. Razmatramo sljedeći niz transformacija:

```
x1, x2 = divide(x, dim=1)
y1 = x1
y2 = x2 + h(x1)
y = concatenate((y1, y2), dim=1)
```

   Varijabla x predstavlja ulazni 4D tenzor oblika (N, C, H, W), gdje je C uvijek paran broj. Funkcija `divide` dijeli zadani tenzor po naznačenoj dimenziji na dva jednaka dijela. Funkcija `concatenate` konkatenira argumente po zadanoj dimenziji.

   Implementirajte navedeni niz transformacija razredom koji nasljeđuje `torch.nn.Module`. Transformaciju h implementirajte kao Conv_1×1 ∘ ReLU. Dozvoljeno je korištenje svih mogućnosti biblioteke Pytorch koje su korištene u laboratorijskoj vježbi.

   Bonus: i) Algebarski odredite inverz navedenog niza transformacija. ii) Vašem razredu dodajte metodu `inverse` koja računa inverz navedenog niza transformacija.

Rješenje zadatka 1.a: Sloj radi s minigrupama vektora $X = [\mathbf{x}_1^T, ..., \mathbf{x}_N^T]^T$, $X_{i:}$ predstavlja i-ti redak matrice X.
Izraz po vektorima:

$$\mathbf{y} = A(\mathbf{x} \odot \mathbf{x}) + b\mathbf{x} + \mathbf{c} = A\mathbf{z} + b\mathbf{x} + \mathbf{c} \tag{1}$$

Uvodimo zamjenu: $\mathbf{x} \odot \mathbf{x} = [x_1^2, ..., x_D^2]^T = \mathbf{z}$
Gubitak po ulazu:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathrm{diag}([2x_1, ..., 2x_D]) = 2\mathrm{diag}(\mathbf{x}) \tag{2}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = A\frac{\partial \mathbf{z}}{\partial \mathbf{x}} + b = 2A\,\mathrm{diag}(\mathbf{x}) + b \tag{3}$$

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\left[\frac{\partial Y}{\partial X}\right]^T \tag{4}$$

Gubitak po parametrima:

$$\frac{\partial y_i}{\partial A_{i:}} = [x_1^2, ..., x_D^2] = \mathbf{x} \odot \mathbf{x} \tag{5}$$

$$\frac{\partial L}{\partial A} = \left[\frac{\partial L}{\partial Y}\right]^T (X \odot X) \tag{6}$$

Rješenje zadatka 1.b:

$$\frac{\partial g(x, \beta)}{\partial x} = \sigma(\beta x) + x \cdot \sigma(\beta x)(1 - \sigma(\beta x))\beta \tag{7}$$

Kod:

```
### 1.a
class Quadratic(Layer):
  # 2 boda
  def __init__(self, input_layer, num_outputs, name,
               weights_initializer_fn=variance_scaling_initializer,
               bias_initializer_fn=zero_init, b):

    self.input_shape = input_layer.shape
    self.N = self.input_shape[0]
    self.shape = (self.N, num_outputs)
    self.num_outputs = num_outputs

    self.num_inputs = 1
    for i in range(1, len(self.input_shape)):
      self.num_inputs *= self.input_shape[i]

    self.A = weights_initializer_fn(
      [num_outputs, self.num_inputs], fan_in=self.num_inputs
    )
    self.c = bias_initializer_fn([num_outputs])
    self.b = b
    self.name = name
    self.has_params = True

  # 2 boda
  def forward(self, inputs):
    self.activations = inputs
    return (inputs * inputs) @ self.A.T + inputs * self.b + self.c

  # 4 boda
  def backward_inputs(self, grads):
    return np.array([
        grads[i]@((2 * self.A * np.diag(self.activations[i])) + b)
        for i in range(N)])
  # 4 boda
  def backward_params(self, grads):
    grad_A = grads.T @ (self.activations ** 2)
    grad_c = grads
    return [[self.A, grad_A], [self.c, grad_c], self.name]

class Swish(Layer):
  # 2 boda
  def __init__(self, name, beta):
    self.name = name
    self.beta = beta
    self.activations = None

  # 2 boda
  def forward(self, inputs):
    self.activations = inputs
    return inputs * self._sigm(self.beta * inputs)

  def _sigm(self, x):
      return 1 / (1 + np.exp(-x))

  # 4 boda
  def backward_inputs(self, grads):
  '''
  d swish(x)/dx = sigm(beta*x) + x*sigm(beta*x)(1 - sigm(beta*x))*beta
  '''
    grad_layer = self._sigm(self.beta * self.activations)
        *(1 + (1 - self._sigm(self.beta * self.activations))
        * self.activations * self.beta)
    return grad_layer.T @ grads

  # 2 boda
  def backward_params(self, grads):
    return None
```

Rješenje zadatka 2.

```
class ConvAvgPool2d(nn.Module):
  # 3 boda
  def __init__(self, channels, K):
    super(ConvAvgPool2d, self).__init__()
    self.K = K
    self.conv = nn.Conv2d(
    channels, channels, kernel_size=K, stride=K, padding=0, bias=False
    )
    self.conv.weight = self.init_weight()

  # 4 boda
  def init_weight(self):
    '''Returns weight tensor of shape (c_out, c_in, K, K)'''
    weight = torch.zeros_like(self.conv.weight)
    for i in range(weight.shape[0]):
      weight[i, i] = torch.ones(self.K, self.K) / (self.K ** 2)
    return weight

  # 1 bod
  def parameters(recurse=True):
    return []

  # 2 boda
  def forward(self, x):
    return self.conv(x)
```

Rješenje zadatka 3.

```
# 1 bod za nasljedivanje nn.Module
class AdditiveCoupling(nn.Module):
  # 3 boda
  def __init__(self, in_channels):
    super(AdditiveCoupling, self).__init__()

    self.relu = nn.ReLU()
    self.conv = nn.Conv2d(in_channels, in_channels, kernel_size=1)

  # 6 bodova
  def forward(self, x):
    x1, x2 = x.chunk(2, dim=1)
    y1 = x1
    y2 = x2 + self.conv(self.relu(x1))
    return torch.cat((y1, y2), dim=1)

  # bonus 6 bodova
  def inverse(self, y):
    y1, y2 = y.chunk(2, dim=1)
    x1 = y1
    x2 = y2 - self.conv(self.relu(x1))
    return torch.cat((x1, x2), dim=1)
```