画像実験課題 A, B

1029323422 天野岳洋

2023 年 1 月 6 日

1 概要

ここでは発展課題 A について取り組んだ内容について述べたのちに, コンテストに対して取り組んだ内容を述べる. 以後明記はしないが, 伝播層では (B, ?, 1) という形で伝播するものとし, 逆伝播層では畳み込み層, プーリング層を除き (?, B) という形でデータを扱っていることに注意したい.

2 AdvancedA

発展課題 A の実装について述べる. 実装が簡単であったり, 定義通りにしか実装していない場合は説明は省略もしくは非常に簡単に述べるものとする.

2.1 A1

活性化関数として Sigmoid 関数の代わりに RELU 関数を用いるという内容である. RELU 関数は次のように表せる関数である.

$$RELU(x) = \begin{cases} x & \text{if } x \geqq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{1}$$

また逆伝播は以下のようである

$$RELU'(x) = \begin{cases} 1 & \text{if } x \geqq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{2}$$

自分の実装を示す.

Listing 1 RELU

```
def __init__(self):
  pass

def prop(self, x):
  self.input = x
  self.B, self.M, _ = x.shape
  return np.where(x <= 0, 0, x)

def back(self, delta):
  return delta * (np.where(self.input <= 0, 0, 1).reshape(self.B, self.M).T
      )
```

2.2 A2

Dropout 層を実装せよという内容である. Dropout 層の定義は以下のものである.

$$Dropout(x) = \begin{cases} x & (\text{ノードが無視されない場合}) \\ 0 & (\text{ノードが無視される場合}) \end{cases} \tag{3}$$

また逆伝播では,

$$\begin{aligned} \frac{En}{x} =& \frac{En}{y}\frac{y}{x} \\ =& \begin{cases} \frac{En}{y} & (\text{ノードが無視されない場合}) \\ 0 & (\text{ノードが無視された場合}) \end{cases} \end{aligned} \tag{4}$$

具体的な実装の説明に移る.Dropout 層のハイパーパラメータを ρ とする. このハイパーパラメータをもとにマスクされるノードの個数を定める. その後 random.choice によってどのノードがマスクされるかを選択し, 適切な処理をすればよい. またテストの際には定義にのっとり, マスクはせずに定数倍を行っている.

Listing 2 Dropout

```
1  def __init__(self, phi, M):
2    self.phi = phi
3    self.msk_num = int(M*phi)
4    self.M = M
5
6  def prop(self, x):
7    B = x.shape[0]
8    M = self.M
9    drop_random = np.repeat(np.random.choice(M, self.msk_num), B).reshape(-1,
         B).T
10   mask_vector = np.ones((B, M))
11   mask_vector[np.repeat(np.arange(B), self.msk_num), drop_random.flatten()]
       = 0
12   self.msk = mask_vector.reshape(B, -1, 1)
13   return x * self.msk
14
15 def back(self, delta):
16   return delta * self.msk.reshape(self.B, self.M).transpose(1, 0)
17
18 def test(self, x):
19   return x * (1 - self.phi)
```

prop 層の mask_vector の作り方が少し複雑なので, 例を挙げて説明する. 例えば, msk_num = 3, M = 5, B = 2 の時を考える. 今 drop_random は 0∼4 から 3 個重複を許さずに選び, それを 2 回

ずつ繰り返し, 次元を (3, 2) に変え転置をとるものだから, 順番に追っていくと, 例えば, 0, 1, 3 が選ばれたとすると, 次のような形で処理が行われ, drop_random が得られることとなる.

$$[0,1,3] \to [0,0,1,1,3,3] \to \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 3 & 3 \end{bmatrix} \to \begin{bmatrix} 0 & 1 & 3 \\ 0 & 1 & 3 \end{bmatrix}$$

続いて同様に mask_vector の推移を説明する. まず, (B, M) の形で全て 1 が入ったもので初期化され, その後 mask_vector[[0,0,0,1,1,1][0 1 3 0 1 3]] = 0 となっている. つまり

$$mask\_vector \to \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \to \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

というような推移になっている. あとはこれを適切な形に変形させ, 入力データとアダマール積をとればあるノードの出力が 0 になっていることが簡単にわかる. また逆伝播層でも入力の shape が違うことを除けば同様にアダマール積をとるだけである.

2.3 A3

Batch-Normalization を行えというものである. 定義式は教科書のとおりである. 実装は定義通り行った. ここでは実際の実装を添付するのみとする.

Listing 3 Batch-Normalization-prop

```
def prop(self, x):
  #x = B * M * 1
  self.x = x
  self.B, self.M, _ = x.shape
  self.microB = np.sum(x, axis=0) / x.shape[0]
  self.sigmaB = np.sum((x - self.microB) ** 2, axis=0) / x.shape[0]
  self.normalize_x = (x - self.microB) / np.sqrt(self.sigmaB +
      BatchNormalize.eps)
  self.y = self.ganma * self.normalize_x + self.beta
  # を yreturn
  return self.y
```

Listing 4 Batch-Normalization-back

```
def back(self, delta):
  delta_xi_head = delta * self.ganma
  delta_sigmaB2 = np.sum(delta_xi_head * (self.x.reshape(self.B, self.M).T -
       self.microB) * (-1/2) * np.power(self.sigmaB + BatchNormalize.eps,
      -3/2), axis=1).reshape(self.M, 1)
  delta_microB = np.sum(delta_xi_head * (-1 / np.power(self.sigmaB +
      BatchNormalize.eps, 1/2)), axis=1).reshape(self.M, 1) + (-2) *
```

```
        delta_sigmaB2 * np.sum(self.x.reshape(self.B, self.M).T - self.microB
        , axis=1).reshape(self.M, 1)
    delta_a = delta_xi_head * np.power(self.sigmaB + BatchNormalize.eps, -1/2)
         + delta_sigmaB2 * 2 * (self.x.reshape(self.B, self.M).T - self.
        microB) / self.B + delta_microB / self.B
    delta_ganma = np.sum(delta * self.normalize_x.reshape(self.B, self.M).T,
        axis=1).reshape(self.M, 1)
    delta_beta = np.sum(delta, axis=1).reshape(self.M, 1)
```

2.4 A4

様々な最適化手法を試すものとなっている. 実際の実装は以下のようである.

```
def adagrad(self, w_b, delta, h):
  h = h + delta * delta
  return w_b - Batch.my * np.power(h, -1/2) * delta, h

def RMSProp(self, w_b, delta, h):
  h = Batch.rho * h + (1 - Batch.rho) * delta * delta
  return w_b - Batch.my * (1 / (np.power(h, 1/2) + Batch.eps) ) * delta, h

def AdaDelta(self, w_b, delta, h, s):
  h = Batch.rho * h + (1 - Batch.rho) * delta * delta
  delta_w = - np.power(s + Batch.eps, 1/2) * np.power(h + Batch.eps, -1/2)
       * delta
  s = Batch.rho * s + (1 - Batch.rho) * delta * delta
  w = w_b + delta_w
  return w, h, s

def Adam(self, w_b, delta, t, m, v):
  t = t + 1
  m = Batch.beta1 * m + ( 1 - Batch.beta1 ) * delta
  v = Batch.beta2 * v + ( 1 - Batch.beta2 ) * delta * delta
  m_head = m / ( 1 - np.power(Batch.beta1, t))
  v_head = v / ( 1 - np.power(Batch.beta2, t))
  W = w_b - Batch.alpha_para * m_head / (np.power(v_head, 1/2) + Batch.eps)
  return W, m, v, t
```

実装の方法は省略するものとし, それぞれの学習方法がどのようなものであるかを考察する.

Momentum 付き SGD

SGD では振動してしまい収束しないという状況が考えられる. Momentum 付き SGD では現在の勾配だけでなく, 過去の差分を用いることによって, この振動を抑えている.

例えば $y = x^2$ のグラフにおいて今の地点が $x = 1$ であり, 学習率が 1 であった場合, 次の地点は $x = -1$ となりさらにその次は $x = 1$ となり振動してしまう. 一方で, Momentum 付き SGD にすることによって, 解消される. 例えば, 0.9 だけ過去の差分を取り入れるとしたら, $x = 1$ の次は, $x = -1$ でありさらにその次は, $\Delta = 0.9 * -2 + 1 * 2 = 0.2$ であるので, $x = -1 + 0.2 = -0.8...$ となり振動が抑えられていることがわかる.

AdaGrad

学習率を調節するものである. SGD の問題として, 収束速度が各軸によってまばらであるという問題がある. 具体的に言うと, ある軸で見たときに勾配が緩やかな状況を考えると, その軸での更新幅は小さく, それ以外の軸での更新幅は大きいということが起こる. すなわち, 学習を進めた際にそれ以外の軸では収束しているのに, その軸が収束していないという状況が考えられる. これを解消するために, 更新が少ない軸では更新幅を大きくするという学習率の面での工夫が AdaGrad である. 教科書の h は今までの更新の 2 乗の総和であり, h が大きい場所程更新幅は小さい。またその逆もしかりである.

RMSProp

AdaGrad では一度大きな更新が行われた軸では学習率が大きく下がってしまったままという問題が発生した. その問題を解決するために, h を今までの単なる総和ではなく, 等比数列的に過去の更新幅を減衰していって加えていくという変更を加えたものである. またこのような処理を加えることによって, h が 0 に収束してしまう場合が考えられるので, 0 割りを防ぐために ϵ が分母に加えられている.

Adadelta

RMSProp 以下では, Δw の次元が w とマッチしていないという事態になっていた. 実際, Δw は基本的に勾配の定数倍であったので, その次元は w の次元を $Dimw$ とすると, $1/Dimw$ となる. これは微分が損失関数の変位 (定数) を w で割ったものの微小極限であることを考えれば明らかである. よって $w + \Delta w$ は次元が違う二つの値を足していることとなる. そのため, なにか不都合な問題が起こった (何が問題なのかを調べきることができませんでした.) これを解決するのが Adadelta である. 実際に次元を見てみると, h の次元は $1/Dimw^2$ であり, s の次元を $Dimw^2$ とすると, Δw の次元は $Dimw * Dimw/Dimw = Dimw$ となり, うまくいっている.[*1]

[*1] s の次元を知るのに, Δw の次元が必要で, その逆も成り立つので, 一度仮定が必要になっています. うまく示せていませんね...

Adam

Momentum + RMSProp である．これは式を見えれば簡単に理解できる．(54), (55) の補正について説明する．(54) の説明だけで十分である．簡易的に一次元値であることを仮定し，さらに β_1 を β としている．教科書式 (52) を変形していく．$\frac{\delta E_n}{\delta W}$ を g とする．

$$\begin{aligned} m_t =&(1-\beta)g_t + \beta m_{t-1} \\ =&(1-\beta)g_t + \beta(1-\beta)(g_{t-1}) + \beta^2 m_{t-2} \\ =&\dots \\ =&(1-\beta)\sum_{i=1}^{t}\beta^{t-i}g_i + \beta^t m_0 \end{aligned}$$

ここで g_t と m_t の平均についての関係性を調べると

$$\begin{aligned} \mathbb{E}[m_t] =&\mathbb{E}[(1-\beta)\sum_{i=1}^{t}\beta^{t-i}g_i + \beta^t m_0] \\ =&(1-\beta)\sum_{i=1}^{t}\beta^{t-i}\mathbb{E}[g_i] + \beta^t\mathbb{E}[m_0] \\ =&(1-\beta)\mathbb{E}[g_t]\frac{1-\beta^t}{1-\beta} \\ =&(1-\beta^t)\mathbb{E}[g_t] \end{aligned}$$

よって不偏性を保つために, 式 (54) によって補正していることが分かった.[*2]

2.5 A5

まだできてません.

2.6 A6

畳み込み層の実装を行う．以下の手順で畳み込み層の実装を行った.

1. 畳み込みフィルタを定義する。初期化は今回正規分布に従う値にした.
2. 入力データを畳み込みフィルタを適用するために、フィルタサイズに合わせて切り分ける。
3. それぞれの切り分けられた部分に対して畳み込みフィルタを適用し、その結果を適切に並べたものを出力とする。
4. バイアスを加える.

特に, 2. が非常にややこしいのでこの部分を説明する.

[*2] そもそもなぜ不偏性が必要なのかがわからなかった. 時間とともに, mt の値が大きくなっていくのが問題?

2.6.1 切り分け

切り分けであるが, 今回は畳み込み前後でサイズが変わらないほうが嬉しいため, フィルタサイズは奇数長であるという制限を設けている. さらにこの切り分けの前に, padding を行っている. padding は numpy.pad で簡単に実装できる. まずは切り分けを行うコードを提示する.

Listing 5 x2X

```
def x2X(self, x, R):
    B, ch, x_length, x_width = x.shape
    dx = x_length - R + 1
    dy = x_width - R + 1
    altx = np.zeros((B, ch, R, R, dx, dy))
    for i in range(R):
        for j in range(R):
            altx[:, :, i, j, :, :] = x[:, :, i:i+dx, j:j+dy]
    return altx.transpose(1, 2, 3, 0, 4, 5).reshape(R*R*ch, dx*dy*B)
```

順に説明を行う. x は切り分けを行う前のテンソルであり, R はフィルタサイズになっている. ここでまず, x のシェイプから各種値を取得している. 次から具体的な切り分けの作業を行っていくが, まず for 文を使う回数をなるべく少なくするために, 大きな四角形でくりぬくことで, 切り分け後の一行一列目の値だけを並べたものを取得できる. このようにすることで繰り返し数をフィルタサイズ * フィルタサイズ程度に収めることができる. 図を使って説明を行う. 以下の図は B = 1, ch = 1 であることに注意したい. 赤四角の大きさが dx, dy である. 左の大きなテンソルの初期化

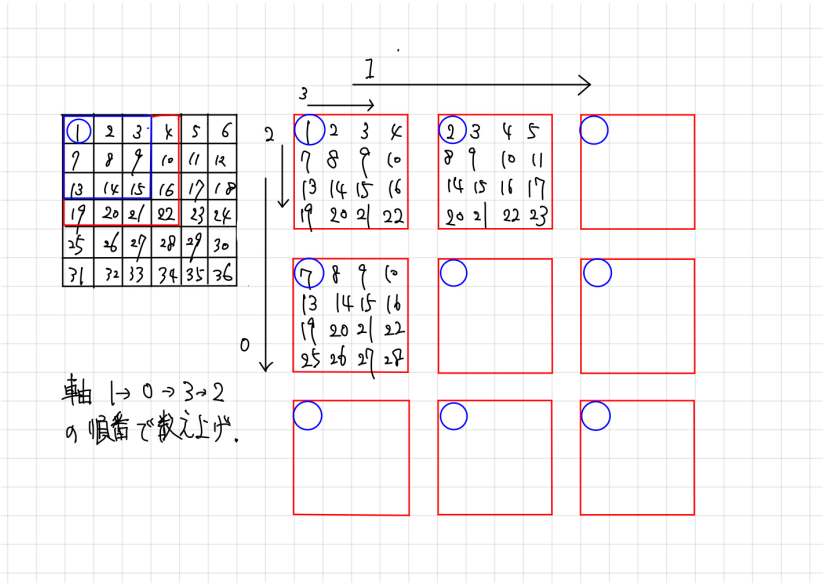

図 1 x2X

を次に行っている. その次の for 文内でそのテンソルにくりぬいている. 最後の transpose が数え

上げの順番を変えている. その後に reshape で適切な形にしている.

適用

切り分けさえできれば後はたやすい. 順伝播は次のようなものになる.

Listing 6　convlution-prop

```
self.B = x.shape[0]
  r = self.R // 2
  self.r = r
  x_prime = np.pad(x, [(0,), (0, ), (r,), (r,)], "constant")
  self.X = self.x2X(x_prime, self.R)
  self.Y = np.dot(self.filter_W, self.X) + self.bias
  # self.Y -> K * (imgsize * B)
  self.Y = self.Y.reshape(self.K, self.B, self.imr, self.imr).transpose(1,
      0, 2, 3)
  # self.Y -> B * K * imlen * imlen
  return self.Y
```

逆伝播

教科書通りの実装である. 気をつけなければならない点として, x2X の逆の作業を行う必要があることと, padding を行っているのでその部分を切り取ったものを出力しなければならないことがあるが, それ以外は簡単である.

Listing 7　X2x

```
def X2x(self, X):
  w = self.R
  bigL = self.imr + self.r*2 - w + 1
  bigW = self.imr + self.r*2 - w + 1
  x = np.zeros((self.B, self.ch, self.imr + self.r * 2, self.imr + self.r
      * 2))
  arX = X.reshape(self.ch, w, w, self.B, bigL, bigW).transpose(3, 0, 1, 2,
       4, 5)
  for i in range(w):
      for j in range(w):
          x[:, :, i:i+bigL, j:j+bigW] = arX[:, :, i, j, :, :]
  return x
```

Listing 8　convlution-back

```
def back(self, delta):
  #delta -> B * K * imlen * imlen
```

```
    delta = delta.transpose(1, 0, 2, 3).reshape(self.K, -1)
    #delta -> K * (B * imlen * imlen)
    delta_filter_x = np.matmul(self.filter_W.T, delta)
    delta_filter_W = np.matmul(delta, self.X.T)
    delta_filter_b = np.sum(delta, axis=0)
    self.filter_W = self.filAdam.update(delta_filter_W)
    self.bias = self.biAdam.update(delta_filter_b)
    return self.X2x(delta_filter_x)[:, :, self.r:self.r + self.imr, self.r:
        self.r + self.imr]
```

2.7 A7

プーリングを行う. 今回は MaxPooling を行うものとする. 以下の手順での実装とする.

1. 切り分けを行い, その窓ごとの最大値とその位置を取得
2. 適切な形に変形
3. 取得した位置をもとに, 逆伝播してきた微分値を代入.
4. x2X の逆の作業によって適切な形に変形

全て以前に出てきたものであるので実装は簡単であるが, 想像がつきづらいので, B = 1, ch = 1, の状況下で具体的に説明を行う. まずは以下の図を見てもらいたい.

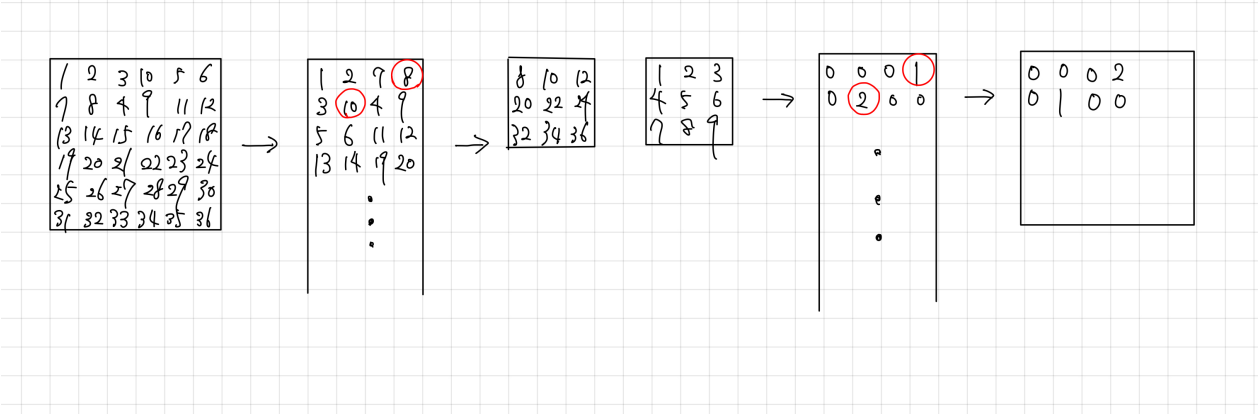

図 2　pooling

まずストライドつきの x2X を行っている. その後 argmax によって, 最大値を取得する位置を取得している. これは逆伝播の際に必要になる. そして最大値を取得して, 適切な形に変形すれば, 順伝播は終了である. 次に逆伝播であるが, これは delta の形が小さくなったもので得られることに注意し, 0 で初期化したものに, 最大値の位置に delta を平坦にしたものを代入すればよい. そしてストライド付きの X2x で適切な形に変形すればよい.

実際の実装は以下のとおりである.

Listing 9　pooling_x2X

```
def x2X(self, x):
  w = self.w
  bigL = self.iml - w + 1
  bigW = self.imw - w + 1
  arrayX = np.zeros((self.B, self.ch, w, w, self.outl, self.outw))
  for i in range(w):
      for j in range(w):
          arrayX[:, :, i, j, :, :] = x[:, :, i:i+bigL:w, j:j+bigW:w]
  X = arrayX.transpose(0, 1, 4, 5, 2, 3).reshape(-1, w*w)
  return X
```

ストライドが追加されていることに注意したい. ストライド幅はプーリングサイズに等しい.

Listing 10　pooling_prop

```
def pooling(self, x, w):
  self.w = w
  self.B, self.ch, self.iml, self.imw = x.shape
  self.outl = self.iml // w
  self.outw = self.imw // w
  X = self.x2X(x)
  self.arg_max = np.argmax(X, axis = 1)
  output1 = np.max(X, axis=1).reshape(
      self.B, self.ch, self.outl, self.outw)
  output2 = output1.transpose(1, 0, 2, 3).reshape(self.B, -1, 1)
  return output1
```

Listing 11　pooling_X2x

```
def X2x(self, X):
  w = self.w
  bigL = self.iml - w + 1
  bigW = self.imw - w + 1
  x = np.zeros((self.B, self.ch, self.iml, self.imw))
  arX = X.reshape(self.B, self.ch, self.outl, self.outw, w, w).transpose(0,
       1, 4, 5, 2, 3)
  for i in range(w):
      for j in range(w):
          x[:, :, i:i+bigL:w, j:j+bigW:w] = arX[:, :, i, j, :, :]
  return x
```

Listing 12　pooling_back

```
1   def back(self, delta):
2     #delta -> B * (ch* imglen)
3     w = self.w
4     delta_x = np.zeros((delta.size, w * w))
5     delta_x[np.arange(self.arg_max.size), self.arg_max.flatten()] = delta.
          flatten()
6     return self.X2x(delta_x)
```

3 contest

コンテストに挑戦した過程を大まかに述べる以下は最終的な構成である.

3.1 Data-argumentation

3.1.1 Affine 変換

3.1.2 gause-noise

3.1.3 thickfiltering

3.2 Convlution

3.3 pooling

3.4

3.5

不採用群

3.5.1 optimaizer:SAM

3.5.2 pseudo-labeling1

3.5.3 randomcrop

3.5.4 randomerasing

3.5.5 mixup

3.5.6 cutout

3.5.7 label-smoothing

4 B

自宅デスクトップでの環境構築から GAN 作成までの手続きを述べる.

4.1 環境

4.2 環境構築

4.3 環境構築 check

Listing 13 check

```
from tensorflow.python.client import device_lib

  print(device_lib.list_local_devices())
```