画像実験課題 A, B

1029323422 天野岳洋

2023 年 1 月 26 日

1 概要

本レポートでは発展課題 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. そして最後に発展課題 B で取り組んだ内容について説明を行う.

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. また, 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.

2.1 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

```
1  def __init__(self):
2    pass
3
4  def prop(self, x):
5    self.input = x
6    self.B, self.M, _ = x.shape
7    return np.where(x <= 0, 0, x)
8
9  def back(self, delta):
10   return delta * (np.where(self.input <= 0, 0, 1).reshape(self.B, self.M).T
         )
```

np.where により, 0 以下を 0 にしてそれ以外を x のままにしている. 逆伝播層では, x にする代わりに 1 にしている. ここで, 行列の形が違うことに注意して変形を行っている.

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,

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

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特定のノードの出力が 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. Batch normalization 層は、ニューラルネットワークの学習において、過学習を防止し、学習速度を上げるために使用される。特に、深いニューラルネットワークでは、層を重ねることで、入力データのパラメータの分布が変化しやすくなり、学習が容易に進まない問題が生じる。これを解消するために、batch normalization 層は、入力データを正規化することで、学習を安定させる効果がある。

具体的に、batch normalization 層では、各ミニバッチの入力データの平均値と標準偏差を計算し、それらを使って入力データを正規化する。また、batch normalization 層はデータの分布が異なることを考えて, 学習時と推論時で別々に平均や標準偏差を計算する。定義式は次のように表せる.

$$\begin{aligned} \mu_B &= \frac{1}{B}\sum_{i=1}^{B} x_i \\ \sigma_B^2 &= \frac{1}{B}\sum_{i=1}^{B}(x_i - \mu_B)^2 \\ \hat{x_i} &= \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ y_i &= \gamma \hat{x_i} + \beta \end{aligned}$$

さらに逆伝播を導出する. $\frac{\partial L}{\partial y_i}$ は与えられていることに注意する.

$$\frac{\partial L}{\partial \hat{x_i}} = \gamma \frac{\partial L}{\partial y_i} \qquad \frac{\partial L}{\partial \gamma} = \frac{\partial L}{\partial y_i}\hat{x_i} \qquad \frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial y_i}$$

次に, $\hat{x_i}$ について

$$\frac{\partial \hat{x_i}}{\partial (x_i - \mu_B)} = \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} \qquad \frac{\partial \hat{x_i}}{\partial \sigma_B^2} = (x_i - \mu_B)\frac{-1}{2}(\sigma_B^2 + \epsilon)^{-\frac{3}{2}}$$

さらに, σ_B^2 について

$$\frac{\partial \sigma_B^2}{\partial (x_i - \mu_B)} = \frac{2}{B}(x_i - \mu_B)$$

また, 自明ではあるが, $x_i - \mu_B$ について

$$\frac{\partial (x_i - \mu_B)}{\partial x_i} = 1 \qquad \frac{\partial (x_i - \mu_B)}{\partial \mu_B} = -1$$

さらに, μ_B について,

$$\frac{\partial \mu_B}{\partial x_i} = \frac{1}{B}$$

以上より次のようにかける.

$$\begin{aligned}
\frac{\partial L}{\partial x_k} &= \sum_{i=1}^{B} \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial x_k} \\
&= \sum_{i=1}^{B} \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial \hat{x_i}}\frac{\partial \hat{x_i}}{\partial x_k} \\
&= \sum_{i=1}^{B}\sum_{l=1}^{B} \frac{\partial L}{\partial y_i}\gamma\left(\frac{\partial \hat{x_i}}{\partial (x_l - \mu_B)} + \frac{\partial \hat{x_i}}{\partial \sigma_B^2}\frac{\partial \sigma_B^2}{\partial (x_l - \mu_B)}\right)\left(\frac{\partial (x_l - \mu_B)}{\partial x_k} + \frac{\partial (x_l - \mu_B)}{\partial \mu_B}\frac{\partial \mu_B}{\partial x_k}\right) \\
&= \sum_{i=1}^{B} \frac{\partial L}{\partial y_i}\gamma\left(\frac{\partial \hat{x_i}}{\partial (x_k - \mu_B)} + \frac{\partial \hat{x_i}}{\partial \sigma_B^2}\frac{\partial \sigma_B^2}{\partial (x_k - \mu_B)}\right) \\
&\quad - \frac{1}{B}\sum_{i=1}^{B}\sum_{l=1}^{B} \frac{\partial L}{\partial y_i}\gamma\left(\frac{\partial \hat{x_i}}{\partial (x_l - \mu_B)} + \frac{\partial \hat{x_i}}{\partial \sigma_B^2}\frac{\partial \sigma_B^2}{\partial (x_l - \mu_B)}\right) \\
&= \sum_{i=1}^{B} \frac{\partial L}{\partial y_i}\gamma\left(\frac{\partial \hat{x_i}}{\partial (x_k - \mu_B)} + \frac{\partial \hat{x_i}}{\partial \sigma_B^2}\frac{\partial \sigma_B^2}{\partial (x_k - \mu_B)}\right) \\
&\quad - \frac{1}{B}\sum_{i=1}^{B} \frac{\partial L}{\partial y_i}\gamma\left(\frac{\partial \hat{x_i}}{\partial (x_i - \mu_B)}\right) \\
&\quad\quad - \frac{1}{B}\sum_{i=1}^{B} \frac{\partial \hat{x_i}}{\partial \sigma_B^2}\sum_{l=1}^{B} \frac{\partial \sigma_B^2}{\partial (x_l - \mu_B)} \\
&= \frac{\partial L}{\partial y_k}\gamma\frac{\partial \hat{x_k}}{\partial (x_k - \mu_B)} + \sum_{i=1}^{B} \frac{\partial L}{\partial y_i}\gamma\frac{\partial \hat{x_i}}{\partial \sigma_B^2}\frac{\partial \sigma_B^2}{\partial (x_k - \mu_B)} \\
&\quad - \frac{1}{B}\sum_{i=1}^{B} \frac{\partial L}{\partial y_i}\gamma\left(\frac{\partial \hat{x_i}}{\partial (x_i - \mu_B)}\right) - \frac{1}{B}\sum_{i=1}^{B} \frac{\partial \hat{x_i}}{\partial \sigma_B^2}\sum_{l=1}^{B} \frac{\partial \sigma_B^2}{\partial (x_l - \mu_B)}
\end{aligned}$$

逆伝播は以上式によって求められる. 教科書との対応だが, [*1]

$$
\begin{aligned}
\frac{\partial E_n}{\partial \hat{x_k}} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} &= \frac{\partial L}{\partial y_k} \gamma \frac{\partial \hat{x_k}}{\partial (x_k - \mu_B)} \\
\frac{\partial E_n}{\partial \sigma_B^2} \cdot \frac{2(x_k - \mu_B)}{B} &= \sum_{i=1}^{B} \frac{\partial L}{\partial y_i} \gamma \frac{\partial \hat{x_i}}{\partial \sigma_B^2} \frac{\partial \sigma_B^2}{\partial (x_k - \mu_B)} \\
\frac{\partial E_n}{\partial \mu_B} \cdot \frac{1}{B} &= -\frac{1}{B} \left(\sum_{i=1}^{B} \frac{\partial L}{\partial y_i} \gamma (\frac{\partial \hat{x_i}}{\partial (x_i - \mu_B)}) + \sum_{i=1}^{B} \frac{\partial \hat{x_i}}{\partial \sigma_B^2} \sum_{l=1}^{B} \frac{\partial \sigma_B^2}{\partial (x_l - \mu_B)} \right)
\end{aligned}
$$

したがって実装は以下のようになる.

Listing 3　Batch-Normalization-prop

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

Listing 4　Batch-Normalization-back

```
1  def back(self, delta):
2    delta_xi_head = delta * self.ganma
3    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)
4    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)
5    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
6    delta_ganma = np.sum(delta * self.normalize_x.reshape(self.B, self.M).T,
         axis=1).reshape(self.M, 1)
7    delta_beta = np.sum(delta, axis=1).reshape(self.M, 1)
```

[*1] 教科書 i をここでは k としています.

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.

```
1  def adagrad(self, w_b, delta, h):
2    h = h + delta * delta
3    return w_b - Batch.my * np.power(h, -1/2) * delta, h
4
5  def RMSProp(self, w_b, delta, h):
6    h = Batch.rho * h + (1 - Batch.rho) * delta * delta
7    return w_b - Batch.my * (1 / (np.power(h, 1/2) + Batch.eps) ) * delta, h
8
9  def AdaDelta(self, w_b, delta, h, s):
10   h = Batch.rho * h + (1 - Batch.rho) * delta * delta
11   delta_w = - np.power(s + Batch.eps, 1/2) * np.power(h + Batch.eps, -1/2)
         * delta
12   s = Batch.rho * s + (1 - Batch.rho) * delta * delta
13   w = w_b + delta_w
14   return w, h, s
15
16 def Adam(self, w_b, delta, t, m, v):
17   t = t + 1
18   m = Batch.beta1 * m + ( 1 - Batch.beta1 ) * delta
19   v = Batch.beta2 * v + ( 1 - Batch.beta2 ) * delta * delta
20   m_head = m / ( 1 - np.power(Batch.beta1, t))
21   v_head = v / ( 1 - np.power(Batch.beta2, t))
22   W = w_b - Batch.alpha_para * m_head / (np.power(v_head, 1/2) + Batch.eps)
23   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.[*2]

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

[*2] 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...

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.[*3]

学習速度の比較

次に以上 6 パターンでの学習速度の比較を行う. 下図左は訓練データに対する, 下図右はテストデータに対するクロスエントロピー誤差を表している. なお最適化に関するハイパーパラメータは教科書にて推奨されているものを使った.

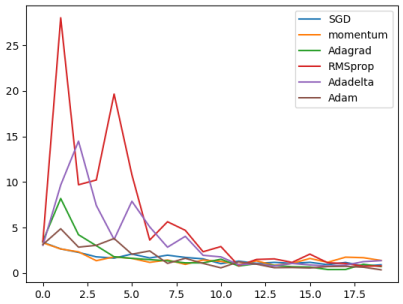

図 1　train_loss

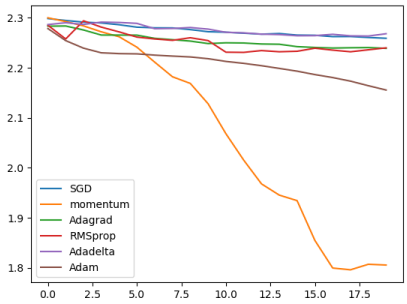

図 2　validation_loss

これを踏まえると, 現在のモデルに対しては, 訓練データ, テストデータともに momentum 付き SGD がもっともよい学習方法であると結論付けることが出来た.

[*3] 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?

2.5 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.

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.

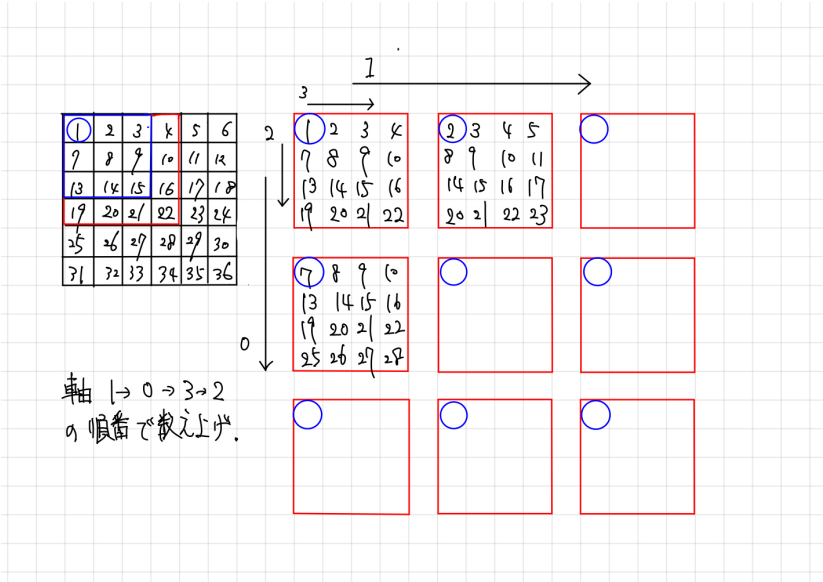

図 3　x2X

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　convolution-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　convolution-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]
```

実際に十分に学習させたフィルタで畳み込みを行うと次のようになる.

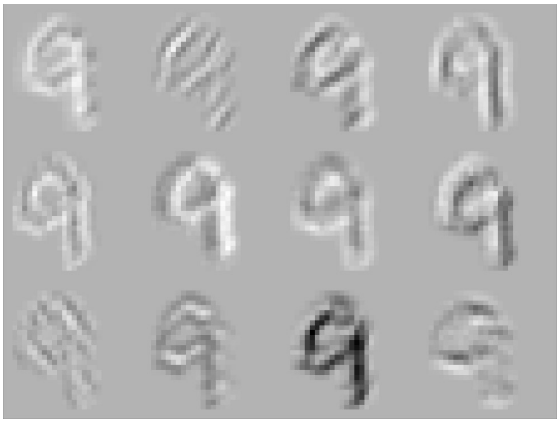

図 4　convolution

2.6 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.

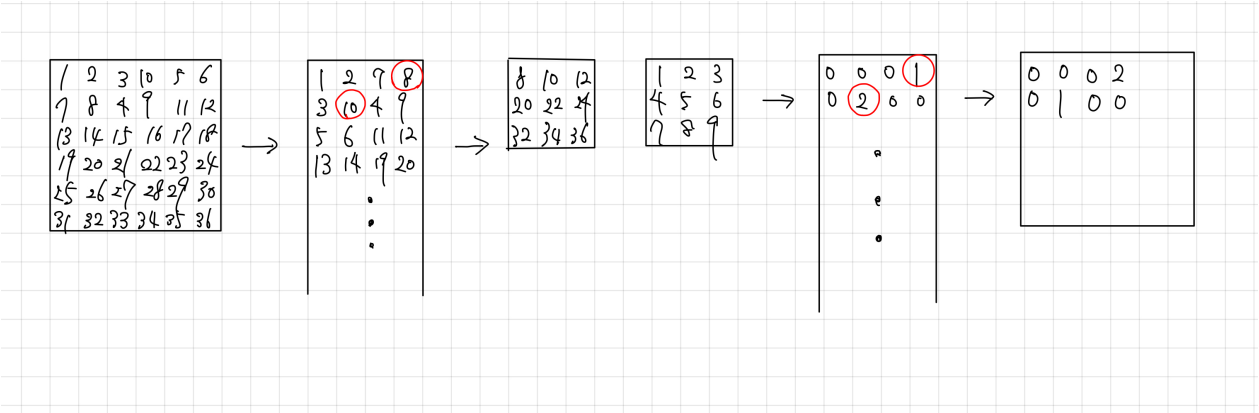

図 5 pooling

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]
```

```
9     X = arrayX.transpose(0, 1, 4, 5, 2, 3).reshape(-1, w*w)
10    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ウィンドウサイズに等しい.

Listing 10　pooling_prop

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

Listing 11　pooling_X2x

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

コンテストに挑戦した. その際にどのような拡張をおこない正答率を高めたかについて説明する. もっともいいパラメータを得ることができたのは, ./AdavancedA/CNN.py である.

3.1 クラス化

試行錯誤の過程で様々な拡張を採用したり, 取り除いたりする必要があるために, Layer や後で説明する拡張を行うものを全てクラス化した. このクラス化したものは./Layer/以下に入っている.

3.2 利便性の向上

学習がいまどれほど進んでいるかを % 表示することや, 途中で打ち切った場合にもそのパラメータを保存することによって, early-stopping を行うことができるようになった.

3.3 model

汎化性能をなるべく高めたモデルにしたい. そこで汎化性能を高める Layer を採用したい. 汎化性能を高めると知られている Layer は Dropout, BatchNormalization, Convolution Layer がある. これを考慮した結果次のような構成になった.

Convolution + MaxPooling

畳み込みを行うことによって, ch を増やし表現力を増やす, 加えて文字認識において重要である周辺情報を利用できるようにする. しかし, これを採用することによって, Dense 層が一つしか使えなくなるので, 線形分離しか表現できないこととなる. このため, MaxPoolingWindow は 4, 7 と大きい幅にして情報を制限しないとうまく分離できなかった. 結果として, 学習速度, 汎化性能を考えたときに ch=32, window=3, poolwindow=7 がちょうどよい結果となった.

BatchNormalization

Convolution の結果に BatchNormalization を行うことによって, 学習の収束速度, 過学習が抑えられる効果があるらしい. 今回はあまり効果を感じなかった. 活性化関数として RELU を用いている.

Dropout

Dropout によって疑似的にアンサンブル学習が行われ, 汎化性能をあげ過学習を抑える効果があると知られているが, 実際過学習を抑える効果があった.

Dense

全結合層, 活性化関数として SoftMax 関数を用いた.

3.4 Data-argumentation

機械学習において精度を高めるために, 最も重要なのは訓練データを増やすことである. しかし, 新たにラベル付けを行うのは時間的コストがかかってしまう. そこで行うのがオーギュメントである. 例えば, 1 の画像を多少平行移動させたり回転させたりしても, それは 1 と認識してほしい. そこでラベルをそのままに, 画像を変形させてあげることで, 訓練データを増やすことができる. 今回ランダムにオーギュメントを行いたいので, バッチごとに逐次オーギュメントを行う (オンラインデータオーギュメント) を行うことにした. contest データを見ながらどのような拡張を行うかを決めた.

Affine 変換

基本的な拡張である Affine 変換である. 同次座標系を用いない疑似的な実装になってしまった. 次の式を見てもらいたい.

$$\begin{pmatrix} y1 \\ y2 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x1 \\ x2 \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

この計算によって, 回転といった変換が可能である. 簡単のために $\begin{pmatrix} e \\ f \end{pmatrix} = 0$ として説明する. 方針としては変換後の (i, j) 画素の位置の変換前の位置を計算する. これは $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ の逆行列を計算し両辺に左から作用させることによって簡単に得ることが出来る.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} x1 \\ x2 \end{pmatrix}$$

一般に (x1, x2) は実数なので, 線形補完によってその画素の値を計算すればよい. これによって (i, j) の画素の値が計算できた. 回転させたければ, 回転角を θ として, $\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$ とすれば回転が実現できる. 下図は mnist のトレインデータのはじめ 4 つに対してそれぞれ $\frac{\pi}{3}, \frac{\pi}{2}$ 回転させたものを並べたものである.

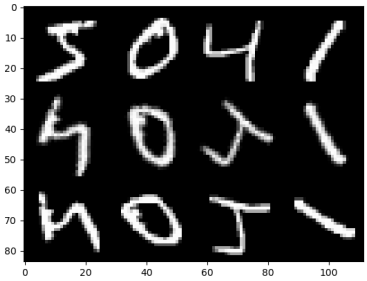

図 6　rotate-image

このようにして, スキュー変換, 平行移動, 回転, 拡大に対する頑健性が得られる. 今回のコンテストで最も正答率を高めることができた効果的な拡張であった. おおよそ 8% の向上がみられた.[*4]

gause-noise

右の図**??**はコンテストデータの 9209 番目のデータである. この画像のようにコンテストデータの中には, 大きくノイズが加わったようなものが見受けられた. そういったノイズに対する頑健性を得れるように, data-argumentation でもノイズを加えるようにする. 今回の実装では, ノイズを平均値 1, 分散 σ^2 の正規分布から (28, 28) の形で生成し, 元の画像とのアダマール積を出力とした. このようにすることによって, 1% ほどの改善が見られた.

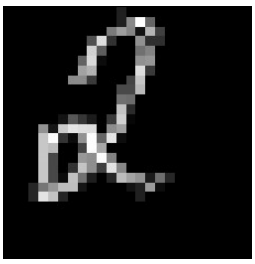

図 7　gause-noise

thickfiltering

同様にコンテストデータの中には, 文字を太くしたような画像が見られたので, それに合わせた data-argumentation を行った. 今回は畳み込みを用いてこれを実装した. 例えば, フィルタを次のようにすると,

$$filter = \begin{pmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 5 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{pmatrix}$$

文字を太くしたような画像が出力される. しかしこのままでは最大値が 255 を上回ってしまうので, 255 以上の値については, 255 にするようにしている. 詳しくは図っていないが, あまり効果は実感できなかった. おそらく Convolution 層で行われていることと, 類似性が高いからであると考えた. 以下の図はこのフィルタリングを行った後の, 数字である. 見てわかるように線が太くなっている.

[*4] はじめ 200 個に対してアノテーションを行い, おおよその正答率を計測しています.

図 8　thickfiltering

3.5　正則化

汎化性能を高める方法として, 正則化が知られている. これは, パラメーターの自由度を下げることによって, 過学習を防ぎ, 汎化性能を高めるというものである. 実装は非常に簡単であり, 損失関数を以下に変更するだけである. X を入力, Y を正解ラベル, W をパラメータとすると,

$$L'(X,Y,W) = L(X,Y) + \lambda d(W)$$

ここで L は通常の損失関数であり, 今回でいうとクロスエントロピーにあたる. そして λ は正則化項をどれくらい効かせるかの定数であり, d は W の重みを表す関数である. 今回は l2 正則化を行うこととした. この時, $d(W)$ は次のように表せる.

$$d(W) = \|W\|_2$$

なので, w_{ij} での偏微分式は次のように表せる.

$$\begin{aligned}\frac{\partial L'(X,Y,W)}{\partial w_{ij}} &= \frac{\partial L(X,Y)}{\partial w_{ij}} + \lambda \frac{\partial d(W)}{\partial w_{ij}} \\ &= \frac{\partial L(X,Y)}{\partial w_{ij}} + 2\lambda w_{ij}\end{aligned}$$

つまり, 全体として更新の際に 2λ 倍した自分自身を加えれば良いことがわかる.

3.6　学習方法

上記の l2 正則化を行うと, Adam よりも Momentum 付き SGD のほうが性能が良かったので, 学習方法としては Momentum 付き SGD を採用した. 詳しい理由についてはわからなかった.

3.7 不採用群

optimaizer:SAM　より平坦なところを目指すようにした optimizer. 理解も浅いまま実装したが, うまくいかず修正方法も思いつかなったため不採用とした.

momentumADAM　始めは Adam で学習して, 後半は momentum 付き SGD で学習することによって, Adam の学習速度の速さと momentum 付き SGD の汎化性能のいいとこどりを試みた optimizer. なぜかうまくいかなかった. Adam と SGD で収束点が全く違うためと予想している.

pseudo-labeling　仮ラベルをつけて学習する. コンテストデータに仮ラベルをつけれればと思ったが pseudo-labeling は仮ラベルを正しくつけれないらしい. 実際うまくいかなかった.

randomcrop　オーギュメントの一種. 画像の一部を切り取る (Dropout の位置関係つきみたいなことをする). おそらく 7 の横棒を切り取ると 1 になるが, ラベルは 7 みたいなことがあるためにうまくいかなかったのではと考察した.

mixup, cutout　オーギュメント. 上と同じ理由でうまく行かなったと考察した.

label-smoothing, softlabeling　6 っぽい 0 と 9 っぽい 0 に同じラベル付けをするのは不自然だと考え, 6 っぽい 0 には 6 の位置にもある程度値を与えたものを onehot-vector とすれば良い (softlabeling). これを実装したかったが, もう一つ CNN を用意してアンサンブルする以外に softlabeling する方法が思いつかなかった. 自分で label を付けてしまうとそれは単に学習を阻害するに終わる. label-smoothing は mnist の label は 100% 正しくて, 誤ラベルがないためうまくいかなかったと考察した.

4 問題点 A

正答率が 94% 付近で停滞してしまった理由について考える. 今回 MaxPooling の windowsize として 7 を選択したが, その場合例えば 9 をプーリングすると下図 5 のようになる. 数字特有の丸

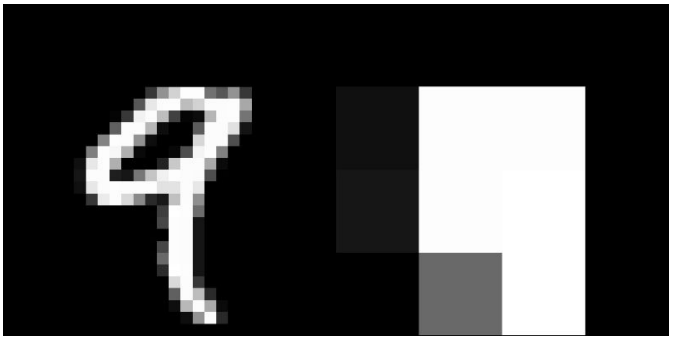

図 9　pooling9

が失われてしまう. さらに Covolution の window サイズは 3 であり, 7 より小さいためこの現象を

防ぐことはできない．このために例えば 9 と 7 の誤認識が非常に多くなってしまった．これを防ぐためには, poolingwindow サイズを小さくすることが考えられるが, それでは線形分離が非常に難しくなってしまい, かえって正答率が悪くなってしまった．これが 3 層 CNN の限界なのではないかと考察する.[*5]

5 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．今回自宅デスクトップの利用なのでサーバーでの実行については想定していない.

5.1 環境

自分の使っている GPU は NVIDIA GeForce GTX 1650 である．まず, NVIDIA の公式サイトから適切な GPU のドライバの最新をインストールした．つぎに, VisualStudio をインストールした．この時 C++ によるデスクトップ開発にチェックを入れなければならないらしい.

次に適切な CUDA, cuDNN をインストールする．おそらく調べたところ windows は全員 GPU によらず, CUDA11.8, cuDNNv8.7.0 でいい？　少なくとも自分は, これでいけました.[*6]cuDNN のファイルを NVIDIA toolkit の適切な場所に配置．(cuDNN の lib の中身は toolkit/version**/lib の中に配置) path の設定．サイトによって path の設定を行ったり, しなかったりとまちまちでしたが, 自分は行いました．CUDA_PATH, CUDNN_PATH をそれぞれ NVIDIA GPU toolkit/cuda/v**に指定しました.

anacondanavigator から新たな仮想環境を追加．root 環境でコンフリクトを起こして, update も downgrade も出来ずに anaconda を再インストールする羽目になったため．その環境に tensorflow-gpu を apply する．version は指定した覚えがないが 2.6.0 が入っていた．[*7]

5.2 環境構築 check

GPU が正しく認識されているかどうかを検査するために以下のコードを実行する.

Listing 13　check

```
1  from tensorflow.python.client import device_lib
2
3    print(device_lib.list_local_devices())
```

これに対する実行結果は,

[*5] keras で同様の構成を行うと, pool_size=(2,2) でもうまくいきました．その時は 96 %程度でした．どこかの実装がうまくいってないのかもしれません.

[*6] cuDNN のインストールに NVIDIA メンバーシップの会員登録が必要

[*7] うまくいかない場合は terminal で conda update conda と conda update anaconda を行えば自動的に anaconda が version を調整してくれるようになるらしいです.

Listing 14　result_check

```
GTX 1650, pci bus id: 0000:09:00.0, compute capability: 7.5
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 5875910335587038350
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 2240439911
locality {
  bus_id: 1
  links {
  }
}
incarnation: 11886741795246851696
physical_device_desc: "device: 0, name: NVIDIA GeForce GTX 1650, pci bus id
    : 0000:09:00.0, compute capability: 7.5"
]
```

また, もっとも簡単な Sequetial モデルによる学習を行うコードは,

Listing 15　check2

```
import numpy as np
import tensorflow.keras as keras
from tensorflow.keras import datasets, models, layers
import tensorflow as tf
import matplotlib.pyplot as plt

physical_devices = tf.config.list_physical_devices('GPU')
if len(physical_devices) > 0:
    for device in physical_devices:
        tf.config.experimental.set_memory_growth(device, True)
        print('{} memory growth: {}'.format(device, tf.config.experimental.
            get_memory_growth(device)))
else:
    print("Not enough GPU hardware devices available")

img_rows, img_cols = 28, 28
num_classes = 10
(X, Y), (Xtest, Ytest) = keras.datasets.mnist.load_data()
X = X.reshape(X.shape[0],img_rows,img_cols,1)
```

```
19  Xtest = Xtest.reshape(Xtest.shape[0],img_rows,img_cols,1)
20  X = X.astype('float32') / 255.0
21  Xtest = Xtest.astype('float32') /255.0
22  input_shape = (img_rows, img_cols, 1)
23  Y = keras.utils.to_categorical(Y, num_classes)
24  Ytest1 = keras.utils.to_categorical(Ytest, num_classes)
25
26
27  model = models.Sequential()
28  model.add(layers.Conv2D(32, kernel_size=(3,3), activation='relu',
        input_shape=input_shape, padding='same'))
29  model.add(layers.Conv2D(64,(3,3),activation='relu',padding='same'))
30  model.add(layers.MaxPooling2D(pool_size=(2,2)))
31  model.add(layers.Flatten())
32  model.add(layers.Dense(128,activation='relu'))
33  model.add(layers.Dense(num_classes, activation='softmax'))
34  print (model.summary())
35
36  model.compile(
37  loss=keras.losses.categorical_crossentropy,
38  optimizer=keras.optimizers.SGD(momentum=0.9), metrics=['acc'])
39
40  epochs = 3
41  batch_size = 32
42
43  result = model.fit(X,Y, batch_size=batch_size,
44  epochs=epochs, validation_data=(Xtest,Ytest1))
45
46  history = result.history
47
48  fig = plt.figure()
49  plt.plot(history['loss'], label='loss')
50  plt.plot(history['val_loss'], label='val_loss')
51  plt.legend()
52  plt.savefig("./AdvanceB/Image/loss_history_withaug.png")
53  fig = plt.figure()
54  plt.plot(history['acc'], label='acc')
55  plt.plot(history['val_acc'], label='val_acc')
56  plt.legend()
57  plt.savefig("./AdvanceB/Image/loss_acc_withaug.png")
```

5.3 B3:GAN

GAN を用いた画像生成器を作れというものが今回の課題の内容である．GAN のアルゴリズムについての説明, 次に実装の方針, 最後に生成された画像とそれに対する考察を行うものとする．

GAN のアルゴリズム

GAN には大きく分けて二つの構成要素が存在する．尤もらしい画像を生成する生成器と生成器によって生成された画像か, それともテストデータかを識別する識別機である．学習の際には, 識別機の学習と, 生成器と識別機を連結させ, 識別機のレイヤーをフリーズしたもの (これを GAN と呼ぶことにする) の学習を交互に行う．

次により具体的な説明を行う． まず生成器は正規分布から得られた形が (100,) のノイズから最終的に (28, 28) の画像を出力する． この次元を増やす作業は, Dense 層と Transposed Convolution 層によって行われる．Transposed Convolution 層で行われていることは単純には Convolution + MaxPooling の逆の動作であり, Upsampling + Convolution によって成り立っている．Upsampling は MaxPooling に対応するもので, 例えば (2, 2) を (4, 4) に拡大する．ここでのアルゴリズムは奇数行奇数列のところにそのまま元の画像を貼り付けて, そして偶数行偶数列のところには左右の値が埋められている所からの線形補間で補われている．このようにすることによって, 100 次元のノイズという圧縮された値から 28*28 の値へと学習可能なパラメータを用いて拡大することができる．そして識別機は CNN を用いることができて, 最後のクラス数が C=2 であること以外は AdvancedA で実装してきたものとほとんど同じである．

また学習であるが, 識別機の学習は今現在の Generator から B/2 枚だけ生成させたものと, Mnist 訓練セットから B/2 枚だけ取得したものとを連結させたものを 1 バッチとして, 識別機に入力として与える．そしてその教師ラベルは $label_x = \begin{cases} [0,1] & (x \in \text{TrainSet}) \\ [1,0] & (x \notin \text{TrainSet}) \end{cases}$ として, 次の式を最小化する．

$$\underset{\theta_d}{\operatorname{argmin}} \sum_{i=0}^{t} label_{x_i} \cdot \log D(x_i)$$

ここで $D(X_i)$ は識別機の出力である．つまり生成器から出力された画像には $[1,0]$ に, 訓練データに対しては $[0,1]$ と予測するように学習させる．これは 2 クラス分類でのクロスエントロピーの最小化と等しい．

次に生成器の学習であるが, 生成器の目的は識別機を騙すことであった．つまり生成器から出力されたものであるが, 識別機が $[0,1]$ と出力させることが目的である．そこで GAN の入力として (B, 100) のノイズを与えて, 教師ラベルをすべて $[0,1]$ として同様にバイナリクロスエントロピーを最小化させてあげればよい．ただしこの時, 識別機のパラメータは更新されないようにする．

しかし, 教科書定義では 2 クラス分類ではなく識別機の出力を入力データが訓練データに含まれている確率を出力していることに注意したい．この点で教科書 GAN の忠実な実装とはなってい

ない.

実装

実装の方針としては, 次の手順に基づいた.

1. generator モデルの生成. ただし, generator 単体で学習させることはないので, コンパイルはする必要はない.
2. discriminator モデルの生成. コンパイルまで行う.
3. discriminator モデルの全ての層をフリーズさせる. ここでフリーズさせたものを反映させるにはコンパイルが必要なので, すでにコンパイルを行っている 2. で得られた discriminator モデルは依然として学習可能である.
4. generator モデルと discriminator モデルを連結してモデル化, その後コンパイルする. これを GAN とする.
5. (discriminator モデルの事前学習)
6. discriminator モデルを 1 バッチ学習させる.
7. GAN モデルを 1 バッチ学習させる.
8. 6. 7. を以降繰り返す.

ここまでは, 比較的簡単に実装できる. GAN の特性として Discriminator と generator がが同等程度の賢さでなければならない. 例えば, discriminator が generator の賢さを大きく上回ってしまうと, generator は学習ができなくなってしまう. [*8] そのため, 学習率といったハイパーパラメータが非常に重要となる. 次のセクションで自分の調整方法を記す.

GAN の学習がうまくいかないとき (工夫点)

GAN を g, discriminator を d と呼ぶことにする. まず, GAN のクロスエントロピーをグラフに出力してみる. その時に滑らかに下がっていることがちょうどいい調整になっている. クロスエントロピーが著しく上昇していく時は, d が強すぎである. 逆にクロスエントロピーがすぐ下がる一方で, 生成される画像は望みのものでない場合は, g が強すぎることが考えられる. ここで自分の場合は d が強くなったので, d を弱くする方法について説明を行う.

学習率　Adam の学習率を下げてやればよい. 自分の場合 d:3*1e-4, g:1e-4 でうまくいった.

label-smoothing　d の学習の際のラベルを曖昧にする. 例えば, [0, 1] を [0.3, 0.7] にする.

Dropout　d に Dropout 層を入れることは, 表現力を落とすことに相当し, 結果的に d を弱くすることができる.

学習回数　d を一回学習させたあと, g を二回学習させるという繰り返しに変更する.(非推奨らしい)

[*8] 実際にその現象は確認できたのですが, discriminator が賢ければ賢いほど generator の学習が進むような気がします. なぜ discriminator が強いとダメになってしまうのかがわかりませんでした.

BatchNormalization　generator には畳み込みや全結合層の後ごとに BatchNormalization 層を挿入する．d への挿入は逆効果の場合あり．

実行結果

ある程度数字らしきものが出力されるようになった．次の画像群はそれぞれ 400 回, 2000 回, 10000 回, 50000 回繰り返した後の同一乱数から出力された画像である．

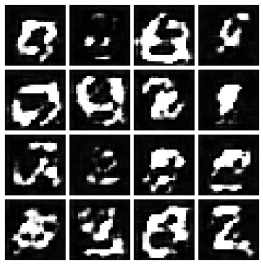

図 10　GAN:400

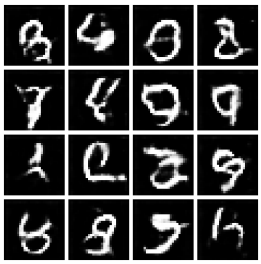

図 11　GAN:2000

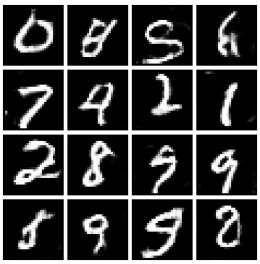

図 12　GAN:10000

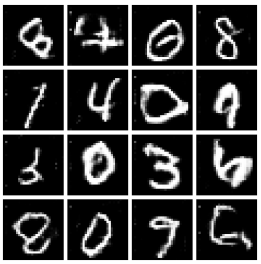

図 13　GAN:50000

問題点 B

generator のロス的には 50000 回学習させた地点のほうが低くなっているが, 上の画像を見ると 10000 回回した所付近が最も自然な文字が出力されていることがわかる．これを踏まえると, earlystopping が必要であると考えられるが, この手法を作ることができていない．そのために, 人力での earlystopping が必要となっている．

他にも, ラベルを活かしていないことも挙げられる．上図を見ればわかるが, 数字らしいものは出力されている一方でそれぞれについて見てみるとどの数字にも分類できないようなものが出力されていることがわかる．これを防ぐために mnist の教師 label が有効なのではないかと考えた．もしこの label を利用することが出来たならば, 少し発展させれば, 出力される数字の値をランダムではなく, たとえば 1 のみを生成することも可能になるだろう．しかしこのラベルの有効な利用には至らなかった.[*9]

[*9] Conditional DCGAN というものを見つけましたが, 実装には至りませんでした.