深層実習
第9,10回

中田尚

再掲

☆課題1

- 2つの正方行列の積を求めるpythonプログラムを作成して、行列サイズと実行時間の関係を調べる
  - 計算ライブラリを使用してはいけません
    - time や matplotlib 以外の import は禁止
    - ライブラリを使った実行は課題2で予定しています
- レポートにはグラフを含めること
  - 単に載せるのではなくグラフから読み取れる内容を説明すること
  - 実行時間の上限は「これ以上増やしても読み取れる内容が増えない」と思えるところまで
    - とはいえ1分以上の実行は不要です
- 締切：4/17 23:59

☆課題2

畳み込みは padding='valid' （小さくなる）で良い
理由を説明すれば 'same' （同じサイズ）でも良い

再掲

- **2つの正方行列の積**と**正方行列と3×3行列の畳み込み**それぞれを求めるpythonプログラムを作成して、それぞれの行列サイズと実行時間の関係を調べる
  - それぞれのpython, numpy, cupy, pytorchの実行時間について考察する
  - cupyの畳み込みは省略しても良い
- レポートにはグラフを含めること
  - 単に載せるのではなくグラフから読み取れる内容を説明すること
  - 実行時間の上限は「これ以上増やしても読み取れる内容が増えない」と思えるところまで
    - とはいえ1分以上の実行は不要です
- 締切： 4/24(木) 23:59

GPUのメモリサイズによっては
10秒程度でメモリ不足になるかも
共有メモリが使われると
一気に遅くなるので注意

それぞれについて「行列サイズと実行時間の間にはXXという関係がある」
という考察が必要

再掲

任意課題1 満点5点

- 課題1の実装は非常に遅く、numpyを使えば100倍以上の高速化が実現できるはずである
- numpyはC言語とFortranで実装されているので、C言語で課題1相当のコードを実装したところ、それでもnumpyの方が10倍程度早い（実行環境によってこの比率は大きく変わる）

- **C言語で実装しただけでは不十分な理由を考えよ**
  - **仮説であれば検証も行うこと（検証方法も自分で考える）**
- **解消可能な理由であれば、自ら実装を行い実行速度がnumpyに近付くことを確かめよ**
  - numpyはGPUを使わないので、GPUは使用禁止
  - cupyとGPU実装で比較しても良い（さらに高難易度）
- 締切：4/24(木) 23:59

再掲

☆課題3

締切
2025/5/15 23:59

- MNISTの学習において、全結合層と畳み込み層の違いが学習時間と精度に与える影響を調査する
  - まず何をどう変更したのかを書く
  - 学習時間と精度のグラフは必須
  - アニメーションは遅いので時間計測時は消すこと
- つまり、
  - mnist.py をダウンロードして実行を確認してから
  - Conv2dを使わないように修正して比較する

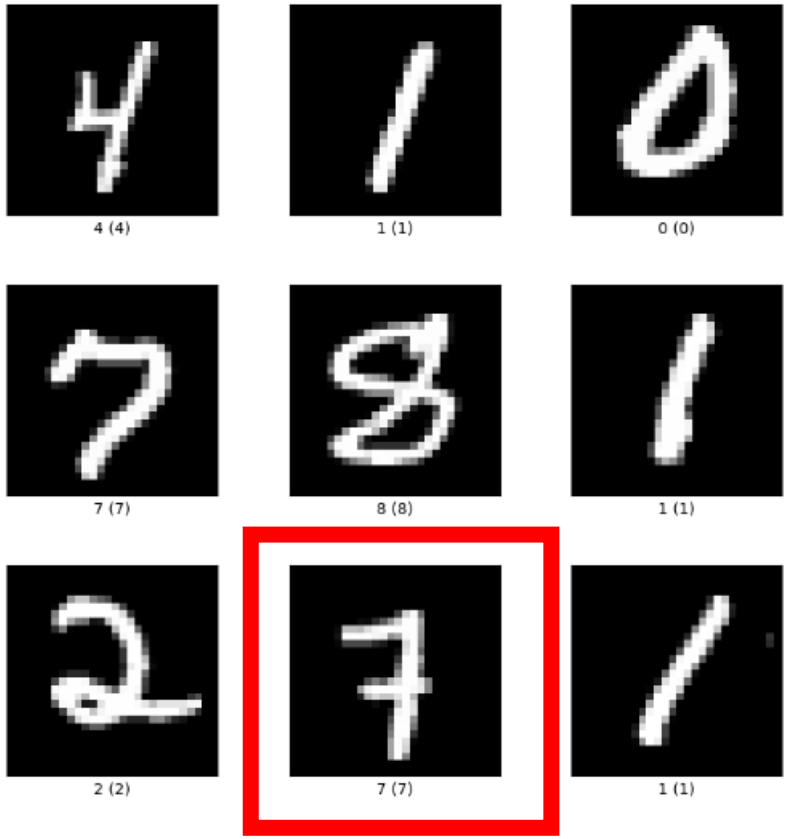
MNISTは簡単なので、「どこまでモデルを小さくして精度が維持できるか、そのためには全結合層と畳み込み層のどちらが良いか」と考えると良い

深層実習
第9,10回

中田尚

MNISTのサンプル

- 日本人にはなじみのない**7**

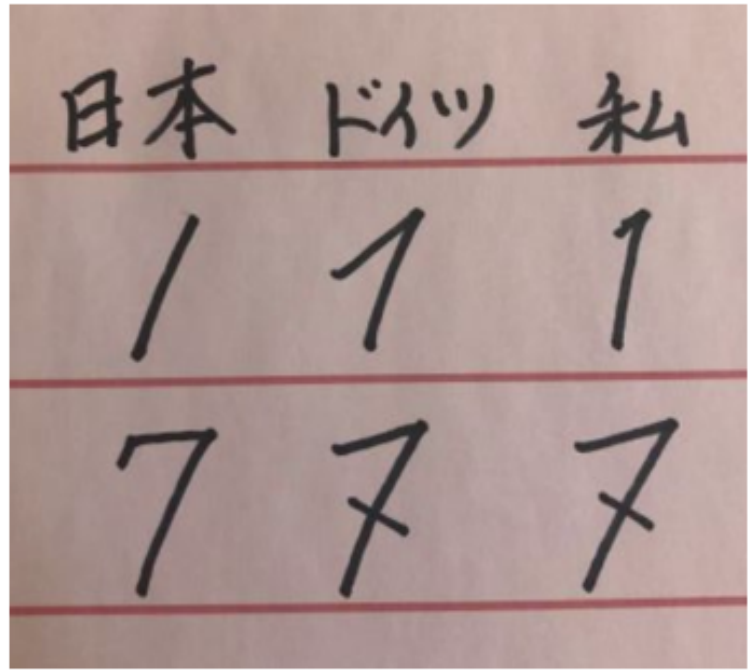


https://www.tensorflow.org/datasets/catalog/mnist

MNISTのサンプル

- 日本人にはなじみのない**7**
- **7**を**7**と手書きで書く国は少ない
  - でもフォントにはある
- **1** が**|**だとアルファベットの**I**（アイ）や**l**（エル）と区別が付かないので**1**を**7**のように書く国は多い
- そのような国では**7**は**7**と書く

- €2.7で**€2.1**出した学生がいた



https://jp.quora.com/%E3%82%A2%E3%83%A9%E3%83%93%E3%82%A2%E6%95%B0%E5%AD%97%E3%81%AE-%EF%BC%91-%E3%82%92%E6%89%8B%E6%9B%B8%E3%81%8D%E3%81%99%E3%82%8B%E3%81%A8%E3%81%8D-%E6%AC%A7%E7%B1%B3%E3%81%AE%E6%96%B9%E3%81%AF-%EF%BC%97-%E3%81%AE

transformについて

- ToTensorで0-255を0-1に変換
- Normalizeでデータの平均を0、標準偏差を1に正規化
  - out=(in-0.1307)/0.3081 を全ピクセルに対して行う

```
# データセットのダウンロードと前処理
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, tra
test_dataset = datasets.MNIST(root='./data', train=False, transform=transfor
print("train=",len(train_dataset))
print("test=",len(test_dataset))
print(train_dataset[0][0].shape,train_dataset[0])

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

端は黒(0)なので、
(0-1307)/0.3081=-0.4242

```
train= 60000
test= 10000
torch.Size([1, 28, 28]) (tensor([[[-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
```

torch.nn （1/2）

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers

Containers

- nn.Sequential

```
model = nn.Sequential(
        nn.Conv2d(1,20,5),nn.ReLU(),
        nn.Conv2d(20,64,5),nn.ReLU())
```

- とすると__init__やforwardすら書かずにmodelを作成できる
  - できるけど、あまり使われない

- 複数のブロックを変数に入れて繰り返し使う場合にも使える

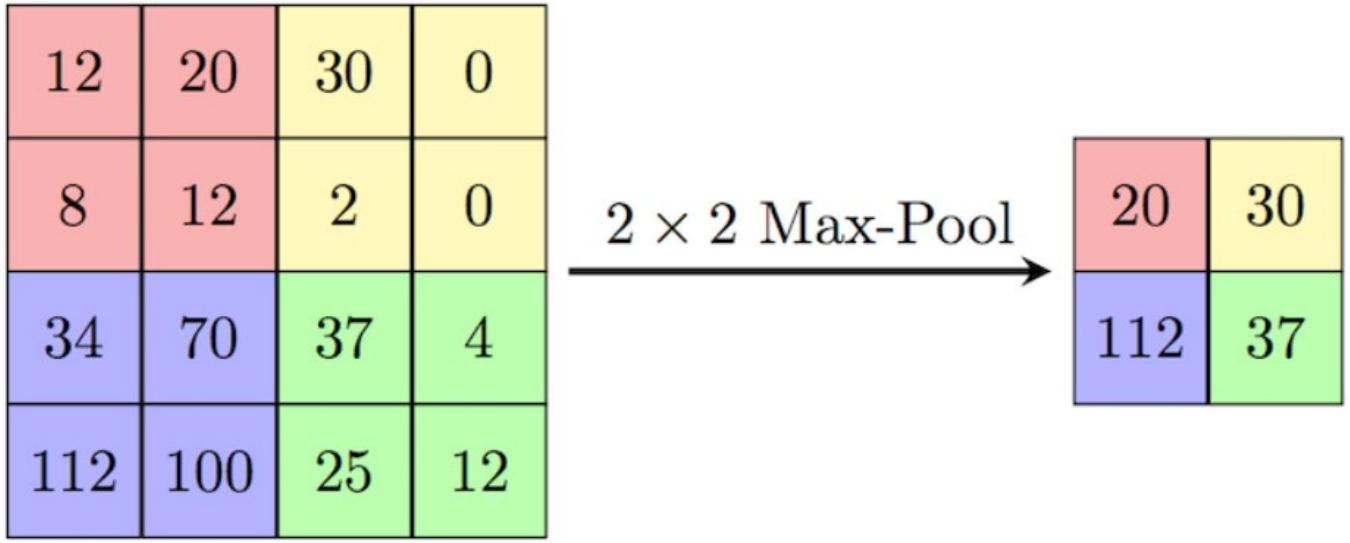
Convolution Layers

- nn.Conv1d
- nn.Conv2d
- nn.Conv3d

- 畳み込み

Pooling layers

- nn.MaxPool1d/2d/3d
- nn.AvgPool1d/2d/3d
- MaxPoolingの方が特徴が保存される（といわれている）
  - 小さい値には情報が少ない



https://computersciencewiki.org/index.php/File:MaxpoolSample2.png

Activations

- Non-linear Activations
  - nn.ReLU
  - nn.Sigmoid
  - nn.Softmax

Recurrent Layers

- nn.RNN
- nn.LSTM

- 再帰的なレイヤ
  - 2年の開発実習で使った

Transformer Layers

- nn.TransformerEncoder
- nn.TransformerDecoder

- 後半で扱いたい

Linear Layers

- nn.Linear

- 全結合層

Dropout Layers

- nn.Dropout
  - 確率的に出力を0にする（学習時のみ）
- Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning
  - ICML'16: Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 June 2016 Pages 1050–1059
- 機械学習の出力の確かさを評価するために、入力や重みにノイズを入れて、出力がどの程度変化するかで評価する手法がある
- Dropoutで近似的に同じ事が出来ると証明した

torch.nn （2/2）

- Sparse Layers
- Distance Functions
- Loss Functions
  - nn.CrossEntropyLoss
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions
- Lazy Modules Initialization

☆課題4

締切
2024/5/22 23:59

- 簡単なニューラルネットワークを使って、（ハイパー）パラメータの違いが学習時間と精度に与える**影響を調査**する
  - パラメータの例
    - ノード数、層数、活性化関数、epoch数、損失関数
  - 基本的には全結合層と畳み込み層で良いが、それ以外を利用しても良い
  - **注目した**パラメータ（**複数可**）と学習時間と精度のグラフは必須
    - **「XXを変えても影響がない」も一つの結果だが、可能な限り「YYは影響がある」を含むと良い**
  - sklearn.datasetsやtorchvision.datasetsのような「ダウンロードするだけのデータセット」**以外**を使うと+1点（つまり9点満点の上限8点）

レポート考察例（チャンネル数）

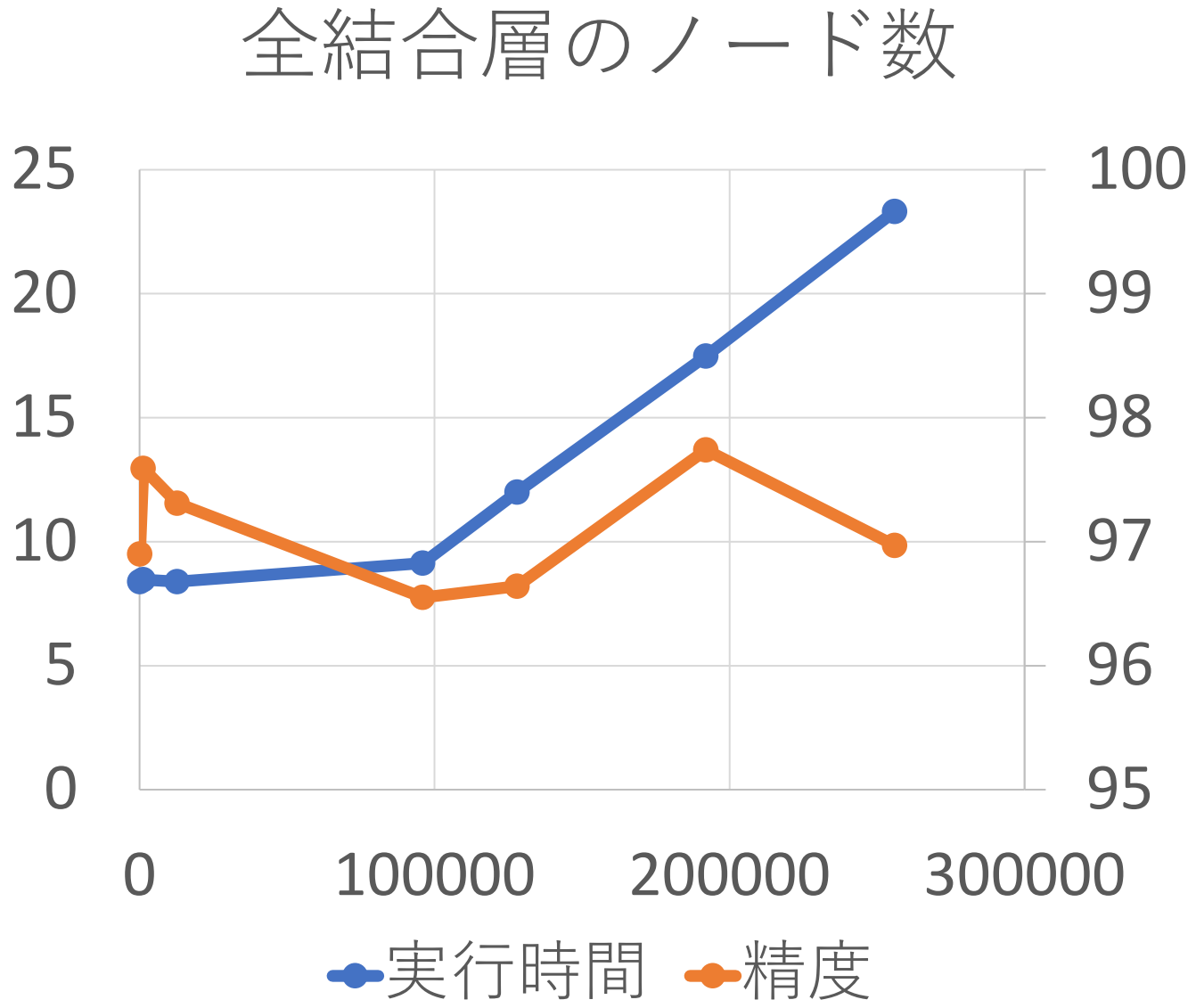
```
super(Net, self).__init__()
self.conv1 = nn.Conv2d(1, 4, 3, 1)
self.conv2 = nn.Conv2d(4, 4, 3, 1)
self.flatten = nn.Flatten()
self.fc1 = nn.Linear(4*24*24, 128)
self.fc2 = nn.Linear(128, 10)
```

レポート考察例（要素数）

```
super(Net, self).__init__()
self.conv1 = nn.Conv2d(1, 4, 3, 1)
self.conv2 = nn.Conv2d(4, 4, 3, 1)
self.flatten = nn.Flatten()
self.fc1 = nn.Linear(4*24*24, 128)
self.fc2 = nn.Linear(128, 10)
```

レポート考察例

- DataLoader等でモデルサイズによらず約8秒かかる



パラメータ数と実行時間

- パラメータ数は大きく違う
- 実行時間はパラメータ数に単純に比例するわけではない
  - なぜ？⇒課題4の追加考察の候補

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
Net                                      [1, 10]                   --
 ├──Conv2d: 1-1                          [1, 4, 26, 26]            40
 ├──Conv2d: 1-2                          [1, 4, 24, 24]            148
 ├──Flatten: 1-3                         [1, 2304]                 --
 ├──Linear: 1-4                          [1, 128]                  295,040
 ├──Linear: 1-5                          [1, 10]                   1,290
==========================================================================
```

任意課題解説

- コンピュータのしくみ
- メモリ
- 並列化

復習：コンピュータの仕組み

- 「命令」に従い「データ」を「演算器」で処理する
  - ノイマン型コンピュータという

- 例
  - ADD a,b,c　a=b+c
  - 1つの「命令」で2つの「データ」を1つの「演算器」で実行

「命令」の高速化

- 演算器を増やす
  - Core i9だとコア当たり3個
  - RTX4090だとコア当たり2個
- コアを増やす（CPUは並列処理が必要）
  - Core i9だと最大18コア
  - RTX4090だと16384コア

- 演算器数x動作周波数で性能の上限が決まる
  - いかに休ませずに使うかが重要

「データ」（アクセス）の高速化

- メインメモリのレイテンシ
  - 要求してからデータが到着するまでの待ち時間
  - 短くても10ns程度
    - 1/1ns=1GHzなので、100MHz程度でしか動作できない

- 解決策1
  - バースト転送
    - 1回に多量のデータを要求する
    - 1回に1000個のデータを要求すると、10ns毎に1000個のデータを受け取れる
      - 実際は1000個同時ではなく、数十個を0.33ns毎に受け取る（3GHz動作の場合）
  - ただしこれはプログラムからは工夫のしようが無い

「データ」（アクセス）の高速化

- 解決策2
  - キャッシュメモリ
    - 1個のデータを要求されても64個のデータを取ってくる
      - 実際は常に64個まとめて取ってくるしか選択肢が無い
    - 例：本を読むときは1ページ毎めくっていく
      - 飛ばすこともあるかもしれないが、連続で読む方が圧倒的に多いはず
      - （もっと細かく言えば、1文字読んだら次の文字を読む）
  - 行列データ（2次元配列）
    - メモリは1次元なので、1ページ1行というイメージ
      - つまり、隣の行は次のページに書いてある

キャッシュメモリ

```
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
    for(k=0;k<N;k++){
      z[i][j]+=x[i][k]*y[k][i]
}}}
```

- x[i][j]はiページ目のj文字目とすると
  - z[1][1]にxの1ページ目1文字目とyの1ページ目1文字目を掛けて足す
  - z[1][1]にxの1ページ目2文字目とyの2ページ目1文字目を掛けて足す

    ↑ここが遅い

- i,j,kの順序は入れ替えても良い（足す順序が変わるだけなので）

  i→k→j の順序にすると
  - z[1][1]にxの1ページ目1文字目とyの1ページ目1文字目を掛けて足す
  - z[1][2]にxの1ページ目1文字目とyの1ページ目2文字目を掛けて足す

  全部1ページ目になった

ブロッキング

```
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
    for(k=0;k<N;k++){
      z[i][j]+=x[i][k]*y[k][i]
}}}
```

- キャッシュのサイズが1ページ分より小さい場合
  - または**1行が複数ページ**と考えても良い←こちらで説明（1ページ100文字）
- i→k→j の順序
  - z[1][100]にxの1ページ目1文字目とyの1ページ目100文字目を掛けて足す
  - z[1][101]にxの1ページ目1文字目とyの2ページ目1文字目を掛けて足す

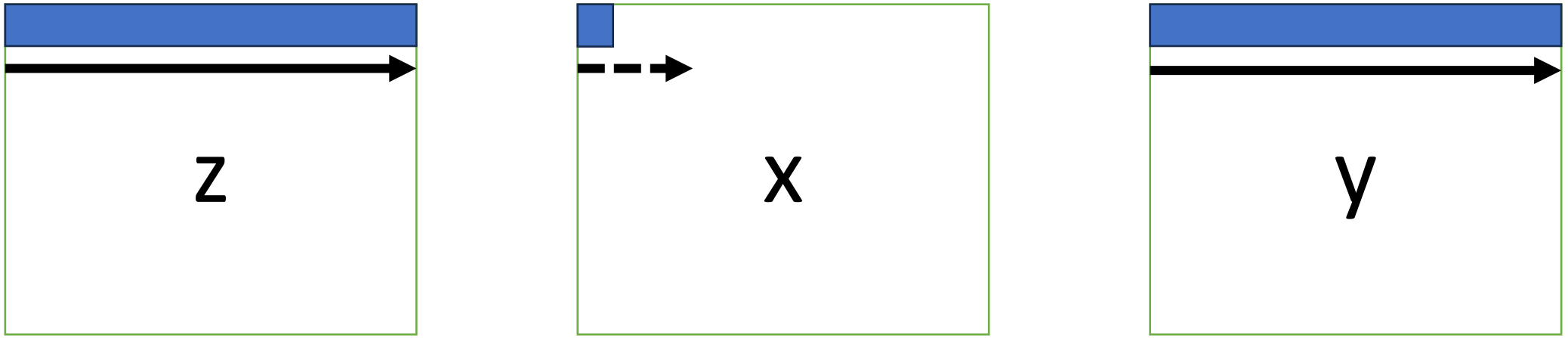
  zとyの1ページ目が消えてしまった

- i→k→j の改良（jが100に到達したら、kを1増やす）
  - z[1][100]にxの1ページ目1文字目とyの1ページ目100文字目を掛けて足す
  - z[1][1]にxの1ページ目2文字目とyの2ページ目1文字目を掛けて足す

  yの1ページ目は消えたが、zは消えなくて済んだ

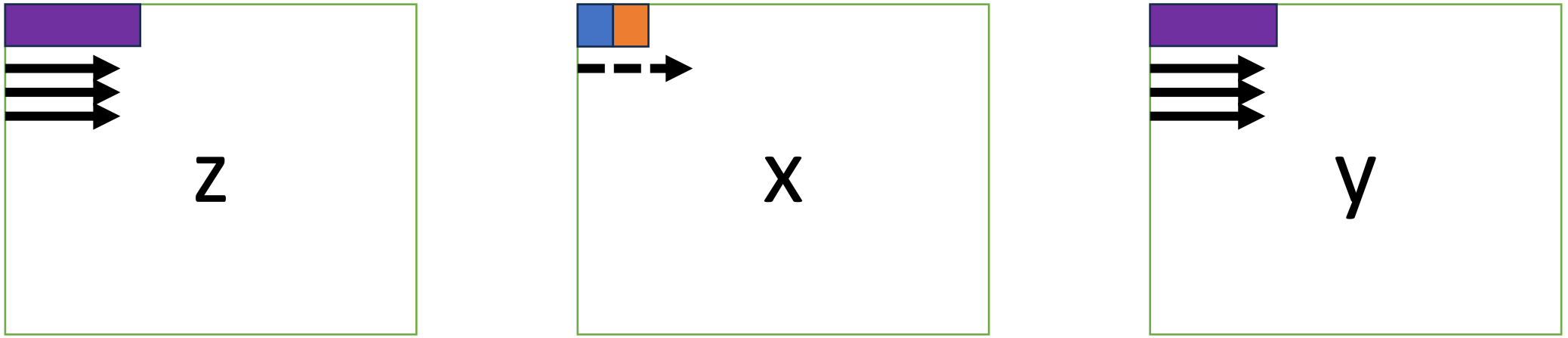
ブロッキングの図解

- ブロッキングなし



ブロッキングの図解

- ブロッキングあり



「命令」の高速化

- プログラムはループで書かれている
  - ADD z[0][0], x[0][0], y[0][0]
  - ADD z[0][1], x[0][0], y[0][1]

  とは（普通は）書けない（行列の足し算に簡略化した）
  - ADD z[i][j], x[i][k], y[k][j]
  - ADD j,j,1 ; j=j+1
  - ADD z[i][j], x[i][k], y[k][j]

  のように間にループ関係の命令が入る
- キーワード
  - アウトオブオーダー（順序を入れ替える）
  - スーパースカラ（複数の命令を同時に実行する）

「命令」の高速化

- 書けないなら書ける命令を作ってしまえ
  - ADD z[0][0], x[0][0], y[0][0]
  - ADD z[0][1], x[0][0], y[0][1]
  - ADD z[0][2], x[0][0], y[0][2]
  - ADD z[0][3], x[0][0], y[0][3]

  を実行する命令を
  - ADD4 z[0][0], x[0][0], y[0][0]

  とする（イメージ）

  - AVX（Intel/AMD）やVFP（ARM）

「全体」の高速化

- 並列化
  - 1台のPCで
    - マルチスレッド、マルチプロセス
    - C言語ならばgcc -fopenmpなどで自動で並列化してくれる
  - 複数台のPCで（Pythorchは非対応）
    - 並列プログラミング（MPIなど）
    - HPF(High Performance Fortran)ではほぼ自動で複数PCで並列化してくれる

「アルゴリズム」の高速化

- そもそも3重ループじゃなくても行列積は計算できる
  - 4x4の行列の掛け算は4^3＝64回の掛け算でできる
  - 49回の掛け算でできる方法が知られていた
    - Volker Strassen, Gaussian elimination is not optimal, Numer. Math. 13 (1969), 354–356.
  - AI(AlphaTensor)で47回の掛け算で済む方法が発見された
    - Fawzi, Alhussein, et al. "Discovering faster matrix multiplication algorithms with reinforcement learning." Nature 610.7930 (2022): 47-53.

（番外編）浮動小数点演算の高速化

- FMA（Fused Multiply-Add）
- 掛け算の総和を求めることが多い
  - 1命令で a+=b*c を実行してしまおう
- 特に浮動小数点演算は演算の前後に前処理と後処理が必要
  - （前処理）（演算）（後処理）（前処理）（演算）（後処理）
  が
  - （前処理）（演算）（演算）（後処理）
  で済むので有利なことが多い
  - 実際の行列積はこんな感じ
    - FMA z[0][0], x[0][0], y[0][0]
    - FMA z[0][1], x[0][0], y[0][1]