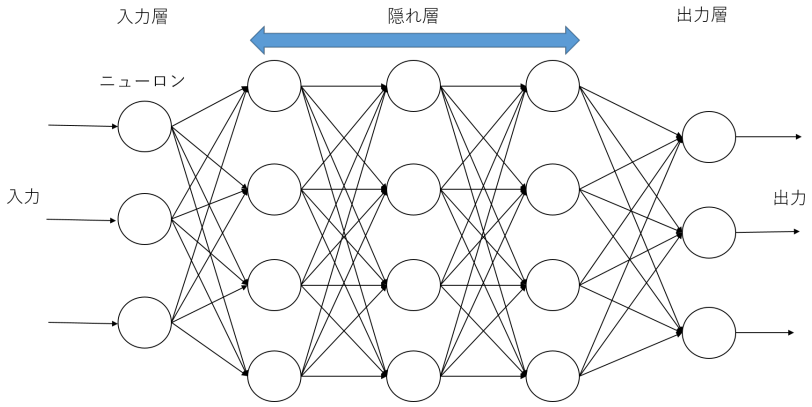
アプリケーション開発　　担当　村松

1　ニューラルネットワーク

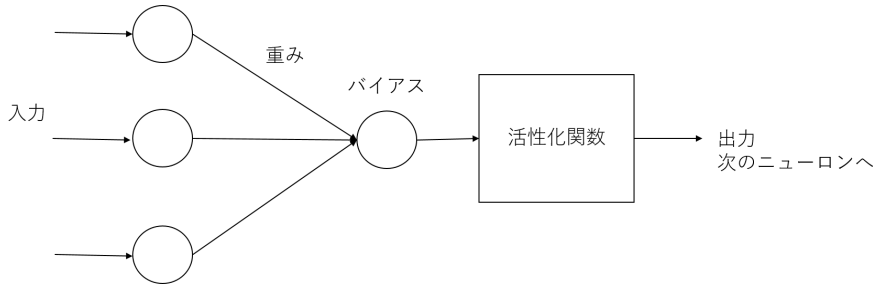
機械学習は下図のようにニューロンと呼ぶ神経細胞を模倣したノードを大量に接続したニューラルネットワークを構築することで種々の問題を解くためのシステムを構築する
問題に応じたデータを与える入力層と回答を行う出力層とその間をつなぐ隠れ層の合計３つの層で構成され、隠れ層の数が多くなるほど複雑な問題を扱うことができるようになる
（ただし層の数が多くなるとパラメータ数が増加し、学習時間が増大する）



ニューロンは下記のように前段の層からのデータに対してある重みを掛け算した値をバイアスとして合計することでデータの統合を行う

$$y = w_1x_1 + w_2x_2 + \cdots + w_nx_n = \sum w_ix_i$$

統合したデータは活性化関数と呼ばれる関数を通じて評価され、次の層のニューロンにデータが渡される



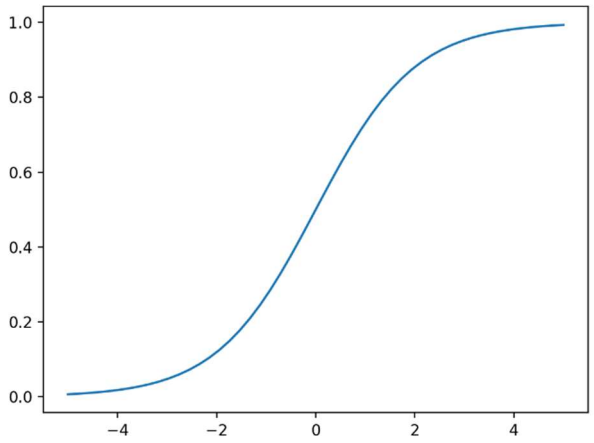
2　活性化関数

活性化関数は次のニューロンにデータを送信するかどうかを判定するための関数であり、動物の脳細胞の機能でいうところの発火を実現するための機能である
ニューロンで統合された単純な積和の数値をこの関数によって非線形な値に変換することで複雑な問題を扱うことが可能となる
活性化関数は様々な種類が研究、提案されおり、ここでは代表的な活性化関数をいくつか紹介する

シグモイド関数
シグモイド関数は 0 から 1 の出力値をとる非線形関数であり関数への入力が小さくなると 0 の値を、大きくなると 1 を返す関数であり下記の数式で定義される

$$y = \frac{1}{1 + \exp(-x)}$$

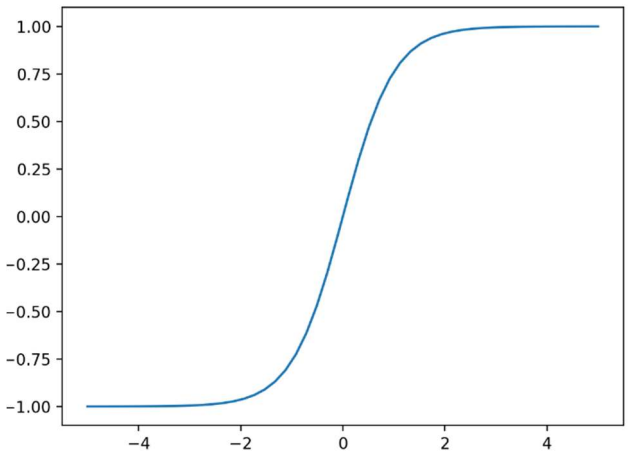
```
import torch
from torch import nn
import matplotlib.pylab as plt

f = nn.Sigmoid() #シグモイド関数

x = torch.linspace(-5,5,50)
y = f(x)

plt.plot(x,y)
plt.show()
```



tanh 関数（ハイパボリック tan 関数）
tanh 関数は-1 から 1 までの出力値をとる非線形関数であり、 0 を中心とした対象性をもつ関数である

$$y = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

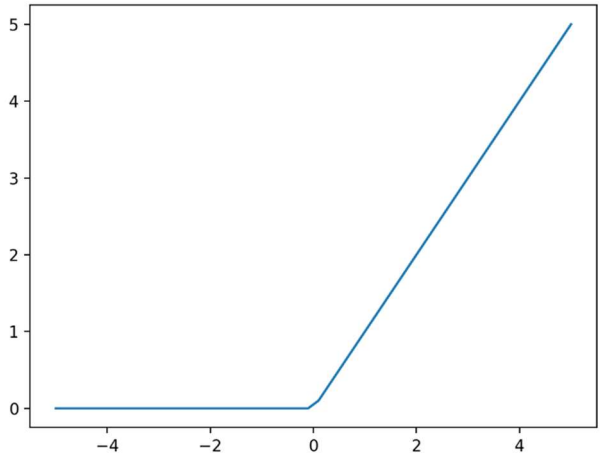
```
import torch
from torch import nn
import matplotlib.pylab as plt

f = nn.Tanh() #Tanh関数

x = torch.linspace(-5,5,50)
y = f(x)

plt.plot(x,y)
plt.show()
```



ReLu 関数（ランプ関数、正規化線形関数）
ランプ関数（もしくは正規化線形関数）は x> 0 の範囲で線形に立ち上がる関数である
1993 年に提案され 2011 年に上記の非線形関数よりもよい性能を得られることが実証され今日広く使われる
一般に活性化関数は非線形のものがよいとされているが、本関数はネットワークのパラメータ調整における逆伝搬における勾配消失問題を緩和することができ、これがネットワークの性能向上につながっていると考察される

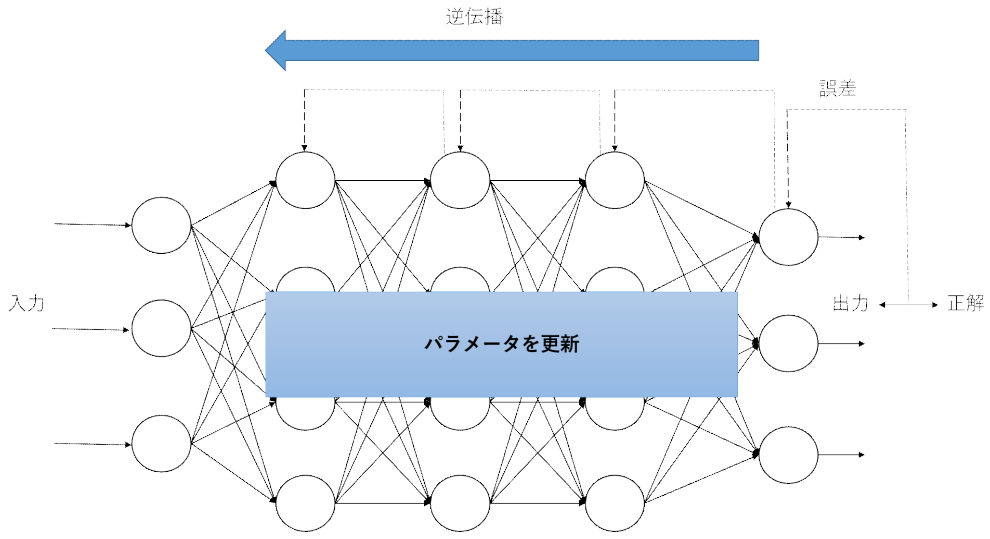
$$y = \begin{cases} 0 & if \quad x \leq 0 \\ x & if \quad X > 0 \end{cases}$$

```
import torch
from torch import nn
import matplotlib.pylab as plt

f = nn.ReLU() #Tanh関数

x = torch.linspace(-5,5,50)
y = f(x)

plt.plot(x,y)
plt.show()
```



3　逆誤差伝搬と損失関数

以上の仕組みで機械学習は入力されたデータから何らかの回答を出力するが最初は適当な回答を出力してしまう
これは各ニューロンで統合される際の重みなどが最適化されておらず適切な値が計算されていないことが原因である
これを解決するためにニューラルネットワークでは出力結果と正解を比較し、その誤差が少なくなるように各層のパラメータを調整することで正解に近づける最適化計算を行う



ここでネットワークと正解がどの程度違っているかを評価するための関数として損失関数と呼ばれる関数を定義する
損失関数にもいくつかの種類の関数があり、扱う問題によって適した関数を利用する

平均二乗誤差関数
平均二乗誤差関数は「回帰問題」という与えられたデータから関数などの数式モデルを予測する問題に適用される損失関数であり、評価項目として予測した関数（出力）と与えられた各データ（正解）とのずれ量の総和を評価することでその誤差を評価するものである（最小二乗法を計算するだけ）
正解をt、出力値をyとし、その偏差を(y - t)とする
この偏差の二乗和を最小化するような関数として定義する

$$\mathbf{E} = \frac{1}{n}\sum_{i=1}^{N}(y_i - t_i)^2$$

```
import torch
from torch import nn

#出力値(ここでは適当な値)
y = torch.tensor([1.0, 1.3, 1.5, 2.1, 3.0, 2.2])

#正解(これも適当な値)
t = torch.tensor([1.3, 1.2, 2.2, 2.4, 2.7, 2.0])

loss_func = nn.MSELoss() #平均二乗誤差
loss = loss_func(y,t)

#損失関数の値を表示
print(loss.item())
```

結果は以下の通り（これは誤差が大きい）

```
==================== REST
0.13500002026557922
>>>
```

交差エントロピー関数
交差エントロピー関数は「分類問題」という与えられたデータがどの属性に所属するのかを予測する（例えば猫の写真を見せたときにどの動物のグループに所属するのか＝猫の画像認識）問題に適用される損失関数であり、あるグループと回答した際の確率分布と正解の確率分布がどの程度一致しているのかということを評価することでその誤差を評価するものである
分類問題は与えられたデータがある属性のものであるということをその確信度で出力する
例えば猫と犬とウマの３つのグループがあり、猫の写真を見せた場合、その正解の確率分布は
（1.0, 0.0, 0.0）となる（猫が 100%、犬が０%、ウマが０%の意）
一方で出力はその確信度となるので、(0,7, 0.2, 0.1) のような形になる（猫 70%、犬 20%、ウマ 10%）

以上より正解の確率分布を t, 出力の確率分布を y とすると、両者の誤差は以下の式で定義される

$$\mathbf{E}(\mathbf{t},\mathbf{y}) = -\sum_{i=1}^{N} t_i \log y_i$$

```
import torch
from torch import nn

#入力値(ここでは適当な値)
x = torch.tensor([[1.0, 2.0, 3.0], #入力1
                  [3.0, 1.0, 2.0]]) #入力2

#正解(これも適当な値)
t = torch.tensor([2,   #入力1に対する正解
                  0]) #入力2に対する正解

loss_func = nn.CrossEntropyLoss() #交差エントロピー関数
loss = loss_func(x,t)

#損失関数の値を表示
print(loss.item())
```

結果は以下のとおり、これも誤差が大きい

```
    ==================== RES
    0.40760600566864014
>>>
```

4　パラメータの更新

損失関数の値を参考に逆誤差伝搬によって各層のパラメータを更新することになるが、闇雲にパラメータの値を変更しても正解にたどり着ける可能性は非常に低いことが容易に想像できる
そこで機械学習ではパラメータを変更した際にどの程度誤差が変化したのかを確認することで効率良くパラメータの調整を行う

ある物理量がどの程度変化したのかという変化量を求めるためには「微分」をすればよい
（例えば距離の変化量は距離を微分した速度で表現され、速度の変化量はこれを微分した加速度で表現されるなど微分は物事の変化量を計算するために用いられる）

損失関数 E をパラメータ w で微分することで調整後の誤差の変化量を求める

$$\text{変化量} \quad = \quad \text{勾配} \quad = \quad \frac{\partial E}{\partial w}$$

SGD （確率的勾配降下法）

学習率ηという係数(学習スピードの司る値)を導入してパラメータ w を以下のように更新する

勾配に係数をかけるだけなので簡単に実装できるが学習の進行度合いで更新量が調整できない

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

```
from torch import optim
optimaizer = optim.SGD(パラメータ,学習率)
```

Momentum

SGD に慣性項 αΔw を追加することで過去の更新量 Δw に応じた最適な更新が実現できる

学習率ηと慣性の影響の強さαと２つパラメータがあるので調整も手間がかかる

$$w \leftarrow w - \eta \frac{\partial E}{\partial w} + \alpha \Delta w$$

```
from torch import optim
optimaizer = optim.SGD(パラメータ,学習率,momentum=XX)
```

AdaGrad

更新量を自動的に調整するために微分値（勾配）の逆数を係数項に取り入れた形で定義したもの

効率のよいパラメータ探索と調整が可能であるが、更新量が減少していく可能性があり更新が進まなくなる場合がある

$$h \leftarrow h + \left(\frac{\partial E}{\partial w}\right)^2 \qquad w \leftarrow w - \eta \frac{1}{\sqrt{h}} \frac{\partial E}{\partial w}$$

```
from torch import optim
optimaizer = optim.Adagrad(パラメータ,学習率)
```

RNSProp

Adagrad の改良版のようなもので係数 p により過去の更新量 h をリセットできるようにしたもの

$$h \leftarrow ph + (1-p)\left(\frac{\partial E}{\partial w}\right)^2 \qquad w \leftarrow w - \eta \frac{1}{\sqrt{h}} \frac{\partial E}{\partial w}$$

```
from torch import optim
optimaizer = optim.RMSprop(パラメータ,学習率)
```

Adam (Adaptive moment estimation)
Momentum と AdaGrad を統合したアルゴリズム
現時点で他の更新則よりも高い性能を発揮するもので広く使われる
パラメータの更新回数 t、任意の定数$\beta_1, \beta_2, \eta, \epsilon$を用いて以下のように定義する

$$m_0 = \quad v_0 = 0$$

$$m_t = \quad \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial E}{\partial w}$$

$$v_t = \quad \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\partial E}{\partial w}\right)^2$$

$$\widehat{m_t} = \frac{m_t}{1-\beta_1}$$

$$\widehat{v_t} = \frac{v_t}{1-\beta_2}$$

$$w \quad \leftarrow \quad w - \eta \frac{1}{\sqrt{\widehat{v_t}}}\frac{\widehat{m_t}}{\epsilon}$$

```
from torch import optim

optimaizer = optim.Adam(パラメータなど)
```

以上、各パラメータの更新則をみれば自明であるが、すべて微分による更新量が含まれている
先に紹介した活性化関数によっては微分値が 1 以下の値をとることになるので、更新が進むごとに勾配が 0 に近づきまったくパラメータの更新が進まない＝学習が進まない事態に陥ることがある
これを勾配消失問題と呼ぶ
ReLu 関数は式より自明であるが微分値が一定となるため、勾配消失が緩和できることが分かる
このため、他の活性化関数よりも良い学習結果が得られる可能性が高いと見ることができる

5 ネットワークの構築

ネットワークを構築するためには nn モジュールに実装されている Sequential の Linear を用いる
nn.Lenear(入力データ次元、出力のニューロン数)

```
import torch
from torch import nn

net = nn.Sequential(
    nn.Linear(64,32),
    nn.ReLU(),
    nn.Linear(32,16),
    nn.ReLU(),
    nn.Linear(16,10))

print(net)
```

出力結果

```
Sequential(
  (0): Linear(in_features=64, out_features=32, bias=True)
  (1): ReLU()
  (2): Linear(in_features=32, out_features=16, bias=True)
  (3): ReLU()
  (4): Linear(in_features=16, out_features=10, bias=True)
)
```

上記は入力層が 64 次元、隠れ層 1 層が 32 次元、隠れ層 2 層が 16 次元、出力層が 10 次元の例

6　簡単な機械学習の流れ

手書き文字の数字を認識する例で機械学習の流れを説明する

まずネット上から学習用データをダウンロードするために Sciklt-learn をインストールする

Windows powershell で以下のように実行

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

新機能と改善のために最新の PowerShell をインストールしてください !https://aka.ms/PSWindows

PS C:\Users\owner> pip3 install scikit-learn
```

まず必要なモジュールをインポートする

```
import torch
from torch import nn
import matplotlib.pyplot as plt
from torch import optim

#Scikit-learnのデータセットモジュール
from sklearn import datasets
#Scikit-learnのデータを学習用とテスト用に分割するモジュール
from sklearn.model_selection import train_test_split
```

Scikit-learn から手書きの数字画像のセットをダウンロードして画像と正解ラベルに分ける

```
#手書きの数字画像セットをダウンロード
digits_data = datasets.load_digits()

#画像セットをdigits_imagesに入力
digits_images = digits_data.data
#正解ラベルをlabelsに入力
labels = digits_data.target
```

データを学習用と学習結果を評価するためのデータに分割する

X_train が学習データ、t_train が学習用の正解ラベル, x_test と t_test が評価用のデータ

慣習的に全データのうちの約 25%くらいを評価用に使用する

```
#データを学習用と評価用に分割
x_train, x_test, t_train, t_test = train_test_split(digits_images,labels)
```

データを pytorch で扱うために tensor に変換する

```
#tensorに変換
x_train = torch.tensor(x_train,dtype=torch.float32)
t_train = torch.tensor(t_train,dtype=torch.int64)
x_test = torch.tensor(x_test,dtype=torch.float32)
t_test = torch.tensor(t_test,dtype=torch.int64)
```

ネットワークを構築する

今回は 8× 8 ピクセルの画像データなので、ピクセルの総数は 64 個

結果は 0 から 9 の 10 個の数字なので、出力結果は 10 次元

よって入力層を 64 次元、出力層を 10 次元とする

隠れ層の構造は適当に 32 次元、16 次元の 2 層とした（実際はちゃんと検討する）

活性化関数は ReLU()とした

```
#ネットワークの構築
net = nn.Sequential(
    nn.Linear(64,32),
    nn.ReLU(),
    nn.Linear(32,16),
    nn.ReLU(),
    nn.Linear(16,10))
```

損失関数と勾配を求めるための最適化アルゴリズムを定義する

分類問題なので損失関数は交差エントロピー関数、最適化は SGD で実装することにする

学習率はとりあえず 0.01 とした(これも実際はちゃんと検討する)

```
#学習
#損失関数の定義
loss_func = nn.CrossEntropyLoss()

#SGDでパラメータ探索
optimizer = optim.SGD(net.parameters(),lr=0.01)
```

学習の様子を確認できるようにログを記録するリストを定義しておく

```
#ログ
record_loss_train = []
record_loss_test = []
```

準備ができたので実際に学習させてみる
とりあえず 1000 回学習させることを試してみる
（＊以下のコードはすべて for i in range(1000)のブロック内に記述すること）

まず勾配を初期化して、適当にデータを与えて出力を得てみる
このように入力を与えてネットワークから出力を得ることを順伝播と呼ぶ
nn.Sequential()で作成したネットワークの変数にデータを与えることで順伝播が計算できる

```
for i in range(1000):
    #パラメータ勾配の初期化
    optimizer.zero_grad()

    #まず適当に出力する（順伝播）
    y_train = net(x_train)
    y_test = net(x_test)
```

順伝搬で得られた結果が正解とどのくらい違うのか損失関数の値を計算して記録する

```
    #学習時のログを記録する
    loss_train = loss_func(y_train, t_train)
    loss_test = loss_func(y_test, t_test)
    record_loss_train.append(loss_train.item())
    record_loss_test.append(loss_test.item())
```

損失関数の値に基づいて勾配（誤差の変化量）を求める＝誤差逆伝播
損失関数の変数に対して backward()を適用することで勾配が計算できる

```
    #逆伝播（勾配を計算）
    loss_train.backward()
```

損失関数から勾配が計算できたら最適化アルゴリズム（今回は SGD）でパラメータを更新する
最適化アルゴリズムを定義した変数に足して step()を適用することでパラメータが更新できる

```
    #パラメータの更新
    optimizer.step()
```

以上で学習のおおまかな流れは完了するが、どのように学習が進んでいるか確認するためにログを表示してみる（これは無くてもよいがあったほうがうまくいっているか確認できて便利である）

```
    #100回学習毎に損失関数の値を表示
    if i % 100 == 0:
        print("Epoch:", i,
              "Loss_train:", loss_train.item(),
              "Loss_test:", loss_test.item())
```

出力結果は以下のような感じ

損失関数の値が順調に減少しているのが確認できる

ただ、損失関数の値が学習完了時もそこそこ高く、まだ改善の余地がありそうであることも見て取れる

```
Epoch: 0 Loss_train: 2.457367420196533 Loss_test: 2.4636590480804443
Epoch: 100 Loss_train: 0.9333822131156921 Loss_test: 0.9381024837493896
Epoch: 200 Loss_train: 0.4229494333267212 Loss_test: 0.42677855491638184
Epoch: 300 Loss_train: 0.2750668227672577 Loss_test: 0.2805684208869934
Epoch: 400 Loss_train: 0.2081855982542038 Loss_test: 0.21881559491157532
Epoch: 500 Loss_train: 0.16838689148426056 Loss_test: 0.18491671979427338
Epoch: 600 Loss_train: 0.14157362282276154 Loss_test: 0.16369014978408813
Epoch: 700 Loss_train: 0.121987484395504 Loss_test: 0.15002089738845825
Epoch: 800 Loss_train: 0.10691017657518387 Loss_test: 0.14025981724262238
Epoch: 900 Loss_train: 0.09490548074245453 Loss_test: 0.13330645859241486
```