

# 12.4 PyTorchでの ニューラルネットワークモデルの構築

本章ではPyTorchの予測モデルを実装

➤ Scikit-learnなどの機械学習ライブラリより少し柔軟である分複雑

# 12.4.1

## PyTorchのニューラルネットワークモジュール

- Torch.nnは複雑なモデルのプロトタイプ化、構築を数行のコードで実行可能
- 基本的な線形回帰モデルを単純なデータセットで訓練
  - このモジュールが何をするのか理解することが目的
- Torch.nnモジュール、torch.optimモジュールの機能を段階的に追加
  - ニューラルネットワークの構築にtorch.nnは使用
  - Torch.nnモジュールを使ってモデル構造を定義しtorch.optimで学習を実行
- Torch.optimはパラメータ更新アルゴリズム(最適化手法)を詰め込んだパッケージ
  - Optimizer.zero\_grad() optimizer.step()等が該当

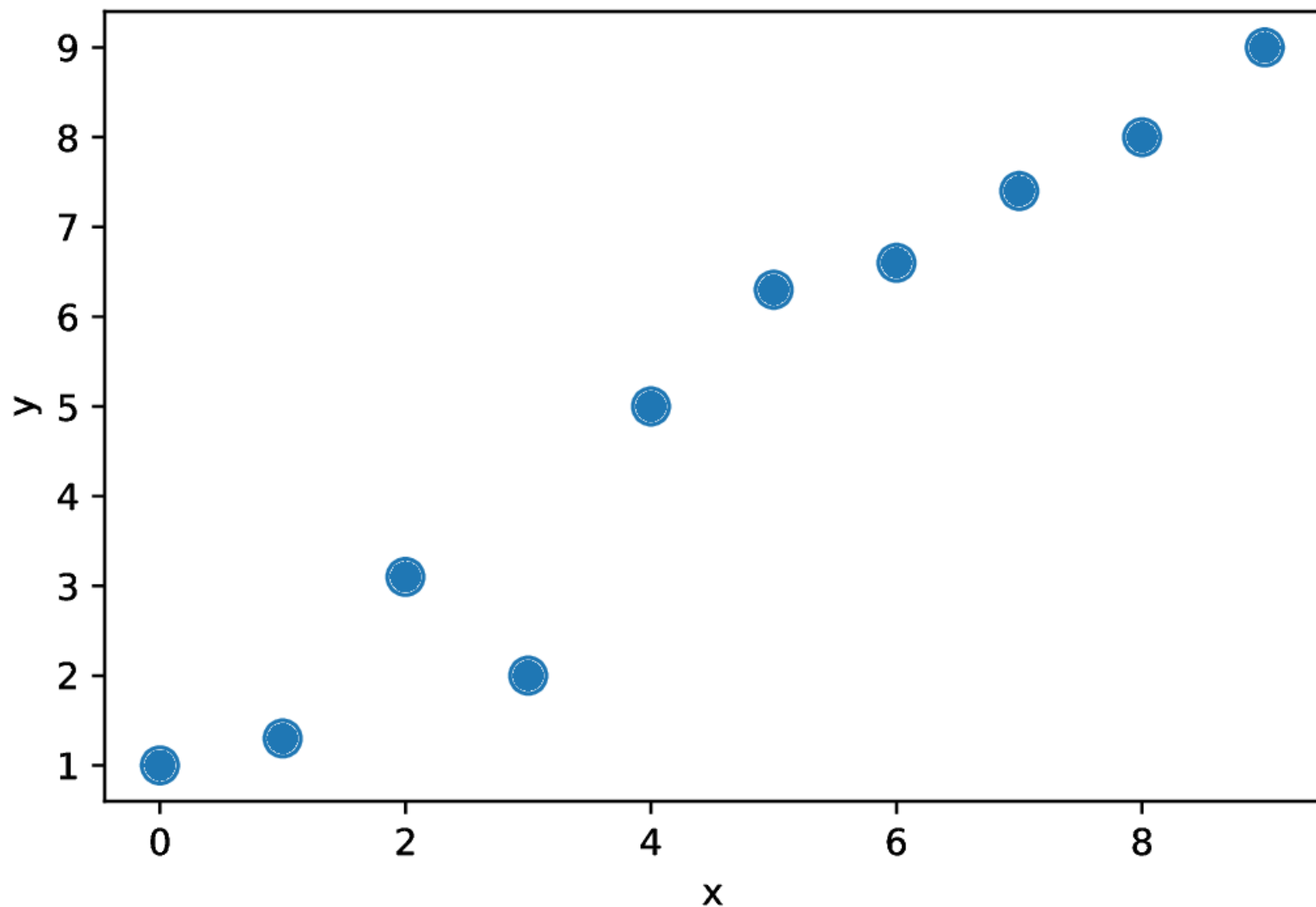
## 12.4.2 線形回帰モデルを構築する

NumPyで単純なデータセットを作成、可視化

```
79     X_train = np.arange(10, dtype='float32').reshape((10, 1))
80     y_train = np.array([1.0, 1.3, 3.1, 2.0, 5.0, 6.3, 6.6,
81                        7.4, 8.0, 9.0], dtype='float32')
82
83     plt.plot(X_train, y_train, 'o', markersize=10)
84     plt.xlabel('x')
85     plt.ylabel('y')
86
87     #plt.savefig('figures/12_07.pdf')
88     plt.show()
```

# NumPyにより可視化されたデータセット

訓練データを散布図として表示



# 線形回帰モデルを構築

- 特徴量を標準化(平均で中心化し、標準偏差で割る)
- 訓練データセットとしてPyTorchのDatasetを作成
- 対応するDataLoaderを作成

```
94     X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
95     X_train_norm = torch.from_numpy(X_train_norm)
96
97     # On some computers the explicit cast to .float() is
98     # necessary
99     y_train = torch.from_numpy(y_train).float()
100
101     train_ds = TensorDataset(X_train_norm, y_train)
102
103     batch_size = 1
104     train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

平均                      標準偏差

# 損失関数を定義

- 損失関数を最小化することでモデルの最適な重みを特定
- 平均二乗誤差(MSE)を損失関数として使用

```
109     torch.manual_seed(1)
110     weight = torch.randn(1)
111     weight.requires_grad_()
112     bias = torch.zeros(1, requires_grad=True)
113
114     def loss_fn(input, target):
115         return (input-target).pow(2).mean()
116
117     def model(xb):
118         return xb @ weight + bias
```

# モデルの重みパラメータの学習

- 確率的勾配法を使用してモデルの訓練を実装
- `Torch.autograd.backward`関数を使用して勾配を計算
- 学習率を設定しモデルを200エポックにわたり訓練

```
120     learning_rate = 0.001
121     num_epochs = 200
122     log_epochs = 10
123
124     for epoch in range(num_epochs):
125         for x_batch, y_batch in train_dl:
126             pred = model(x_batch)
127             loss = loss_fn(pred, y_batch)
128             loss.backward()
129
130             with torch.no_grad():
131                 weight -= weight.grad * learning_rate
132                 bias -= bias.grad * learning_rate
133                 weight.grad.zero_()
134                 bias.grad.zero_()
135
136         if epoch % log_epochs == 0:
137             print(f'Epoch {epoch} Loss {loss.item():.4f}')
```

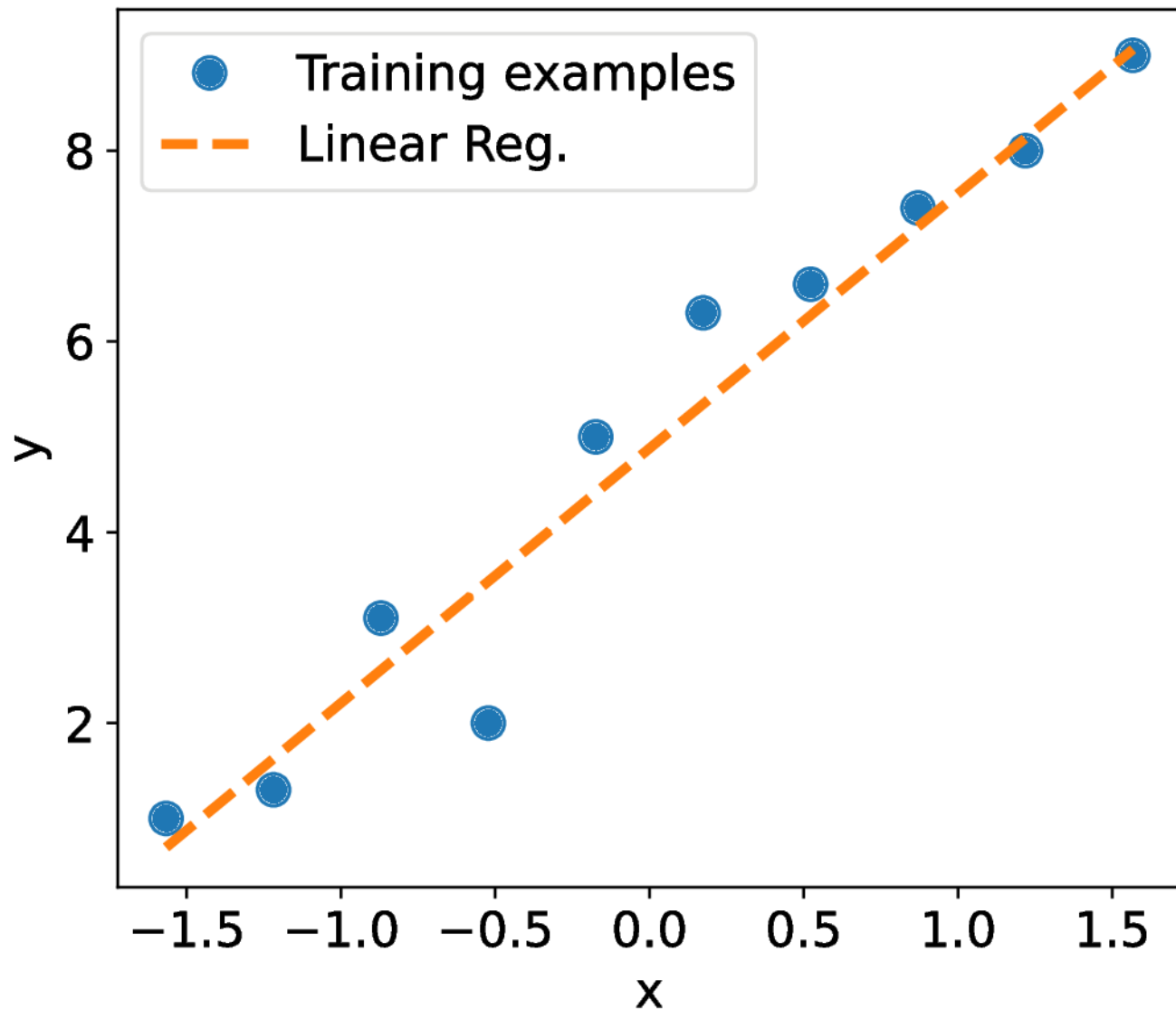
# 訓練したモデルをプロット

- 0～9の値が等間隔に並んだNumPy配列を作成
- モデルの訓練と同様にテストデータにも同じ標準化を適用

```
143     print('Final Parameters:', weight.item(), bias.item())
144
145     X_test = np.linspace(0, 9, num=100, dtype='float32').reshape(-1, 1)
146     X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
147     X_test_norm = torch.from_numpy(X_test_norm)
148     y_pred = model(X_test_norm).detach().numpy()
149
150
151     fig = plt.figure(figsize=(13, 5))
152     ax = fig.add_subplot(1, 2, 1)
153     plt.plot(X_train_norm, y_train, 'o', markersize=10)
154     plt.plot(X_test_norm, y_pred, '--', lw=3)
155     plt.legend(['Training examples', 'Linear Reg.'], fontsize=15)
156     ax.set_xlabel('x', size=15)
157     ax.set_ylabel('y', size=15)
158     ax.tick_params(axis='both', which='major', labelsize=15)
159
160     #plt.savefig('figures/12_08.pdf')
161
162     plt.show()
```

# 訓練データと訓練した線形回帰モデルの散布図

- 0～9の値が等間隔に並んだNumPy配列を作成
- モデルの訓練と同様にテストデータにも同じ標準化を適用



## 12.4.3

# torch.nnとtorch.optimを使ってモデルを訓練

- MSE損失関数と確率的勾配降下法オプティマイザを作成
  - 勾配降下法は損失が大きいほうと逆を選択
- 線形層を明示的に定義する代わりにtorch.nn.Linearクラスを使用

```
170     input_size = 1
171     output_size = 1
172     model = nn.Linear(input_size, output_size)
173
174     loss_fn = nn.MSELoss(reduction='mean')
175
176     optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

# 訓練したモデルの結果

- 前項の手法と同じような結果になることを確認
- 重みパラメータ
- バイアスパラメータ
- モデルの訓練はこの3メソッドを用いる
  - この訓練にはバッチバージョンのデータセット(前項のtrain\_dなど)を渡すことが可能

```
178     for epoch in range(num_epochs):
179         for x_batch, y_batch in train_dl:
180             # 1. Generate predictions
181             pred = model(x_batch)[: , 0]
182
183             # 2. Calculate loss
184             loss = loss_fn(pred, y_batch)
185
186             # 3. Compute gradients
187             loss.backward()
188
189             # 4. Update parameters using gradients
190             optimizer.step()
191
192             # 5. Reset the gradients to zero
193             optimizer.zero_grad()
194
195             if epoch % log_epochs==0:
196                 print(f'Epoch {epoch} Loss {loss.item():.4f}')
197
201     print('Final Parameters:', model.weight.item(), model.bias.item())
```

エポック回数

順伝搬、予測

損失計算

逆伝搬で勾配を計算

パラメータ更新

勾配リセット

## 12.4.4 Irisデータセットの花の品種を 分類する多層パーセプトロンを構築する

- PyTorchのtorch.nnモジュールはニューラルネットワークの構成要素として利用できる層を定義済
- torch.nnモジュールとIrisデータセットを使って分類問題を解く方法を確認
  - 3種類のアヤメの花を識別
- torch.nnモジュールを使用して2層のパーセプトロンを構築
  - sklearn.datasetsからデータを取得
- 訓練データ100個、テストデータ50個

```
228     iris = load_iris()
229     X = iris['data']
230     y = iris['target']
231
232     X_train, X_test, y_train, y_test = train_test_split(
233         X, y, test_size=1./3, random_state=1)
```

# モデルを効率よく構築する準備

- 特徴量を標準化(平均で中心化、標準偏差で割る)
- 訓練データセットとしてPyTorchのDatasetを作成
- 対応するDataLoaderを作成

```
239     X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
240     X_train_norm = torch.from_numpy(X_train_norm).float()
241     y_train = torch.from_numpy(y_train)
242
243     train_ds = TensorDataset(X_train_norm, y_train)
244
245     torch.manual_seed(1)
246     batch_size = 2
247     train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

- nn.Moduleクラスを使って複数の層を積み上げてニューラルネットワークを構築
- 分類問題にはLinear層を使用
  - 全結合層または線形層とも呼ばれる  
 $f(w \cdot x + b)$ と表現可能
- xは入力特徴量を含むテンソル
- wは重み行列
- bはバイアスベクトル
- fは活性化関数
- シグモイド活性化関数
- ソフトマックス活性化関数

```
252  class Model(nn.Module):
253      def __init__(self, input_size, hidden_size, output_size):
254          super().__init__()
255          self.layer1 = nn.Linear(input_size, hidden_size)
256          self.layer2 = nn.Linear(hidden_size, output_size)
257
258      def forward(self, x):
259          x = self.layer1(x)
260          x = nn.Sigmoid()(x)
261          x = self.layer2(x)
262          x = nn.Softmax(dim=1)(x)
263          return x
264
265      input_size = X_train_norm.shape[1]
266      hidden_size = 16
267      output_size = 3
268
269      model = Model(input_size, hidden_size, output_size)
```

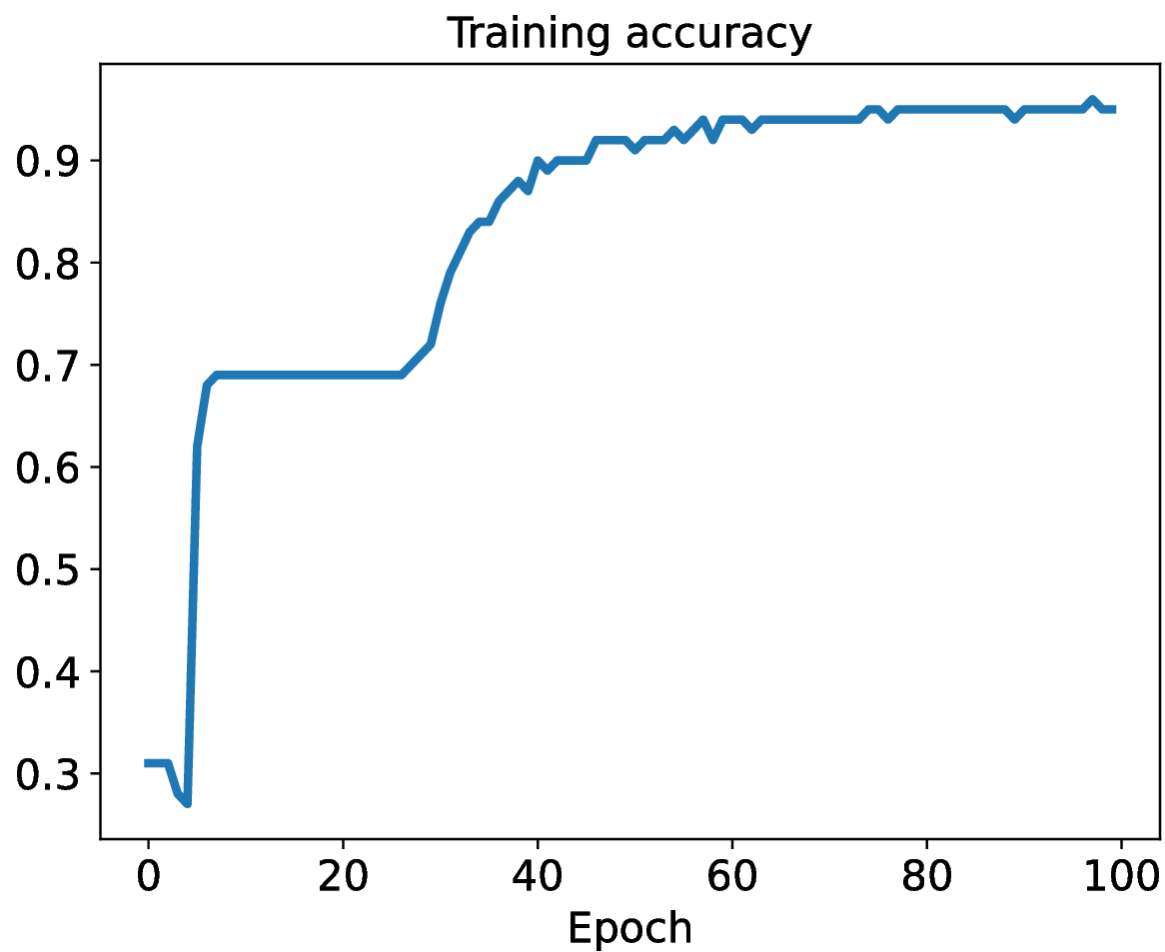
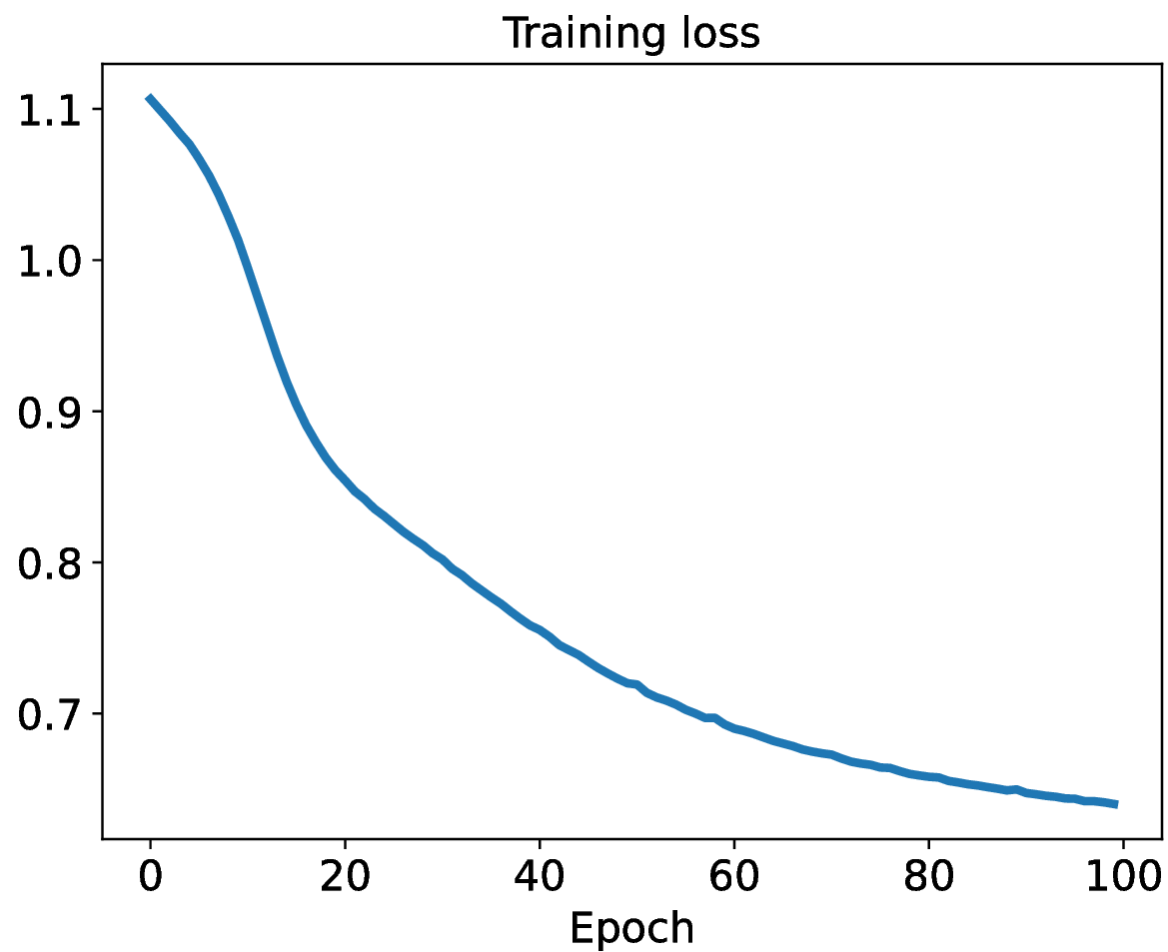
- 損失関数として交差エントロピー損失関数
  - 正解クラスの確率が高いほど損失が減少
- オプティマイザとしてAdamを指定
  - Adamオプティマイザは勾配ベースの堅牢な最適化法
  - 自動で学習率を調節

```
271     learning_rate = 0.001
272
273     loss_fn = nn.CrossEntropyLoss()
274
275     optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

- エポック数100
- loss\_hist: 各エポックの損失値
- accuracy\_hist: 各エポックの正解率

```
280     num_epochs = 100
281     loss_hist = [0] * num_epochs
282     accuracy_hist = [0] * num_epochs
283
284     for epoch in range(num_epochs):
285
286         for x_batch, y_batch in train_dl:
287             pred = model(x_batch)
288             loss = loss_fn(pred, y_batch.long())
289             loss.backward()
290             optimizer.step()
291             optimizer.zero_grad()
292
293             loss_hist[epoch] += loss.item()*y_batch.size(0)
294             is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
295             accuracy_hist[epoch] += is_correct.sum()
296
297             loss_hist[epoch] /= len(train_dl.dataset)
298             accuracy_hist[epoch] /= len(train_dl.dataset)
```

# 学習曲線（訓練の損失率と正解率）



## 12.4.5

# 訓練したモデルをテストデータセットで評価する

- 訓練したモデルをテストデータセットで評価
- モデルの訓練と同様に標準化をテストデータに適用
- 分類正解率は0.98(98%)

```
326     X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
327     X_test_norm = torch.from_numpy(X_test_norm).float()
328     y_test = torch.from_numpy(y_test)
329     pred_test = model(X_test_norm)
330
331     correct = (torch.argmax(pred_test, dim=1) == y_test).float()
332     accuracy = correct.mean()
333
334     print(f'Test Acc.: {accuracy:.4f}')
335
336
337     # ### Saving and reloading the trained model
```

## 12.4.6 訓練したモデルの保存と読み込み

- `save()`はモデルのアーキテクチャと学習したパラメータの両方を保存
- `model_new.eval()`でモデルのアーキテクチャを確認

```
341     path = 'iris_classifier.pt'
342     torch.save(model, path)
347     model_new = torch.load(path)
348     model_new.eval()
```

## 12.5 多層ニューラルネットワークでの 活性化関数の選択

- 本書ではS字型のロジスティック関数を機械学習の文献と同じようにシグモイド関数と定義
- (多層ニューラルネットワークでは)微分可能な関数は活性化関数として扱うことが可能
- 隠れ層の活性化関数として双曲線正接関数が頻出
  - シグモイド関数が0に近い出力を返す場合学習に時間がかかり、極小値に陥る可能性が高くなるため

## 12.5.1 ロジスティック関数のまとめ

- 2値分類タスクにおいて陽性クラスに分類される確立をモデル化

- シグモイド関数

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- 総入力 $z$ の定義

$$z = \mathbf{w}^T \mathbf{x} + b$$

```
381     X = np.array([1, 1.4, 2.5]) ## first value must be 1
382     w = np.array([0.4, 0.3, 0.5])
383
384     def net_input(X, w):
385         return np.dot(X, w)
386
387     def logistic(z):
388         return 1.0 / (1.0 + np.exp(-z))
389
390     def logistic_activation(X, w):
391         z = net_input(X, w)
392         return logistic(z)
393
394     print(f'P(y=1|x) = {logistic_activation(X, w):.3f}')
```

# 複数のロジスティック活性化ユニット

- 3クラス問題の所属確率としては解釈不可能
  - 値の合計が1でない
- クラスラベルの予測が可能
  - 最大値を与えるクラスラベルを予測値と設定

実行結果

Net Input:

[1.78 0.76 1.65]

Output Units:

[0.85569687 0.68135373 0.83889105]

```
399     # W : array with shape = (n_output_units, n_hidden_units+1)
400     # note that the first column are the bias units
401
402     W = np.array([[1.1, 1.2, 0.8, 0.4],
403                  [0.2, 0.4, 1.0, 0.2],
404                  [0.6, 1.5, 1.2, 0.7]])
405
406     # A : data array with shape = (n_hidden_units + 1, n_samples)
407     # note that the first column of this array must be 1
408
409     A = np.array([[1, 0.1, 0.4, 0.6]])
410     Z = np.dot(W, A[0])
411     y_probab = logistic(Z)
412     print('Net Input: \n', Z)
413
414     print('Output Units:\n', y_probab)
```

# 12.5.2 softmax関数を使って 多クラス分類の所属確率を推定する

## softmax関数

- argmax関数のソフトバージョン
- 所属確率を算出
  - 多クラス分類問題においてクラスの所属確率を計算可能
- 各クラスの所属確率の合計が1
  - 分母に正規化項
- 総入力のデータ点がi番目に分類される確率を計算可能

```
427     def softmax(z):  
428         return np.exp(z) / np.sum(np.exp(z))  
429  
430     y_probab = softmax(Z)  
431     print('Probabilities:\n', y_probab)  
432  
433     np.sum(y_probab)  
438     torch.softmax(torch.from_numpy(Z), dim=0)
```

## 12.5.3

# 双曲線正接関数を使って出力範囲を拡大する

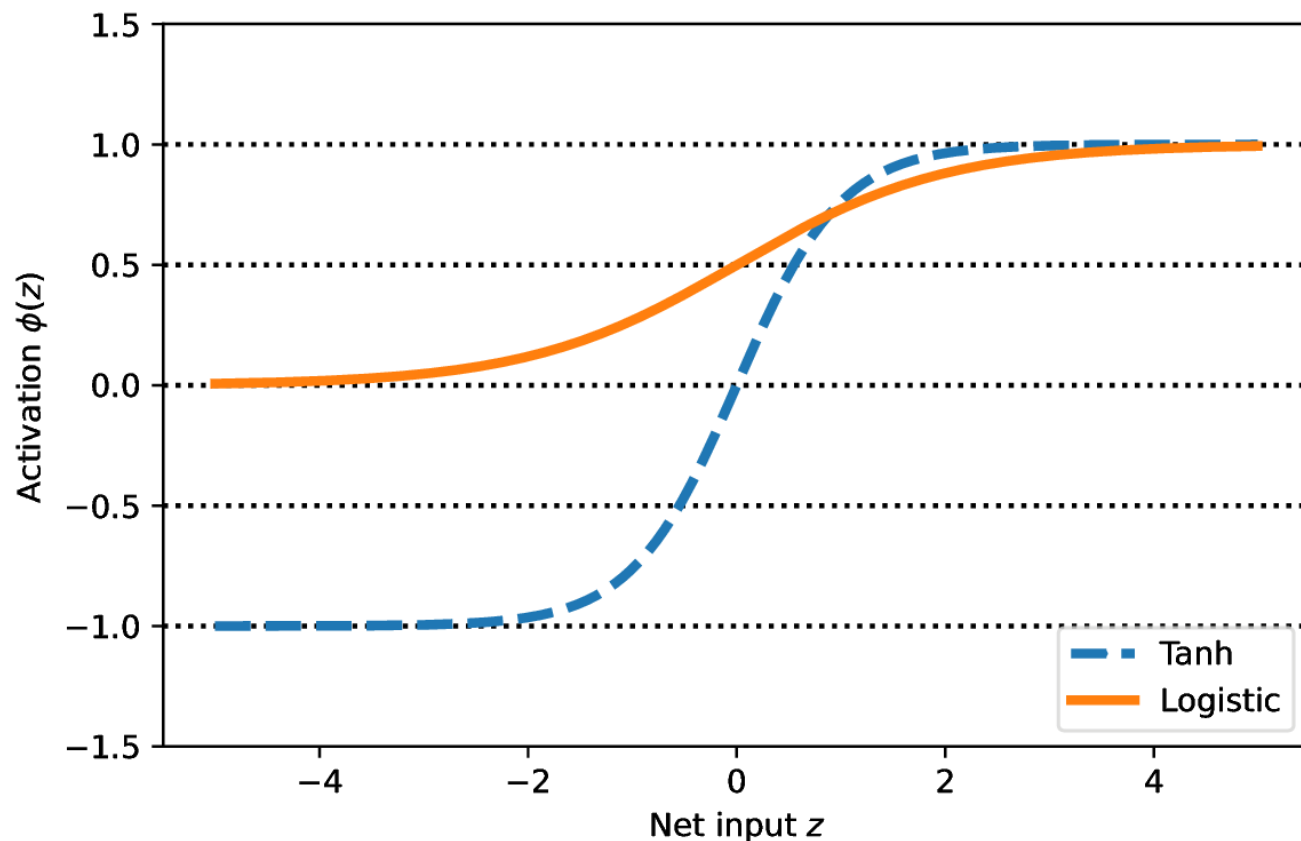
### 双曲線正接関数

- ロジスティック関数の尺度を取り直したバージョン
- ロジスティック関数と比較して出力範囲が広く  
開区間  $(-1, 1)$  におよぶ
- 誤差逆伝播法の収束を改善可能

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

# シグモイド関数のプロット

- tanh関数の大きさはlogistic関数の2倍
- tanh関数(-1,1)、logistic関数(0,1)



```
446 def tanh(z):
447     e_p = np.exp(z)
448     e_m = np.exp(-z)
449     return (e_p - e_m) / (e_p + e_m)
450
451 z = np.arange(-5, 5, 0.005)
452 log_act = logistic(z)
453 tanh_act = tanh(z)
454 plt.ylim([-1.5, 1.5])
455 plt.xlabel('Net input $z$')
456 plt.ylabel('Activation $\phi(z)$')
457 plt.axhline(1, color='black', linestyle=':')
458 plt.axhline(0.5, color='black', linestyle=':')
459 plt.axhline(0, color='black', linestyle=':')
460 plt.axhline(-0.5, color='black', linestyle=':')
461 plt.axhline(-1, color='black', linestyle=':')
462 plt.plot(z, tanh_act,
463         linewidth=3, linestyle='--',
464         label='Tanh')
465 plt.plot(z, log_act,
466         linewidth=3,
467         label='Logistic')
468 plt.legend(loc='lower right')
469 plt.tight_layout()
470
471 #plt.savefig('figures/12_10.pdf')
472 plt.show()
```

## 12.5.4 ReLU活性化関数

- 双曲線関数とロジスティック関数には勾配消失問題が存在
  - 総入力に関して活性化関数の微分は $z$ が大きくなると消失してしまう  
 $z_1=20 \Rightarrow z_2=25$   
 $\sigma(z_1) = 1.0 \Rightarrow \sigma(z_2) = 1.0$
- 活性化関数  $\sigma(z)=\max(0,z)$ 
  - 非線形関数
  - ReLUの微分は正の入力値について常に1になることで勾配消失問題を解決
  - ディープニューラルネットワークに適している

# まとめ

- PyTorchの基本的なテンソル機能を使ってモデルを1から定義
- torch.nnモジュールを利用することでニューラルネットワークモデルの定義を簡略化
  - 1から実装するには行列とベクトルの乗算レベルでのプログラミングと演算の定義が必要
- 双曲線正接関数
  - Sigmoidより中心が0になるので学習が安定
  - 大きな入力では勾配消失が発生
- ソフトマックス関数
  - 出力を確立として扱う
  - 出力の合計1
- ReLU
  - 勾配が1のため勾配消失を防止
  - zが常に負である場合そのノードは更新されなくなる