

POLITECHNIKA RZESZOWSKA  
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI



WYDZIAŁ  
**ELEKTROTECHNIKI  
I INFORMATYKI**  
POLITECHNIKI RZESZOWSKIEJ

**Sztuczna Inteligencja**  
**Klasyfikator win w Burn**  
Projekt

## Spis treści

1. Opis problemu .....	3
1.1. Istota problemu .....	3
1.2. Technologiczne aspekty rozwiązania .....	3
1.3. Cel projektu .....	4
2. Opis części praktycznej .....	4
2.1. Uczenie modelu .....	4
2.2. Nadzorowane uczenie .....	4
2.3. Ocena skuteczności modelu .....	5
2.3.1. Dokładność .....	5
2.3.2. Strata .....	5
3. Analiza danych wejściowych .....	6
3.1. Opis zbioru danych .....	6
3.2. Struktura zbioru danych .....	6
4. Przygotowanie danych do nauki .....	7
4.1. Czyszczenie danych .....	7
4.2. Normalizacja danych .....	8
4.3. Podział na zbiory treningowe i testowe .....	9
4.4. Zapis danych do plików .....	10
4.5. Podsumowanie .....	10
5. Trenowanie modelu .....	11
5.1. Przygotowanie danych .....	11
5.1.1. Implementacja tensorów .....	11
5.2. Definicja modelu .....	13
5.3. Trenowanie modelu .....	14
5.3.1. Wyniki trenowania .....	15
6. Eksperymenty .....	17
6.1. Generowanie danych .....	17
6.2. Zmiana hiperparametrów .....	17
6.3. Zmiana architektury modelu .....	19
7. Podsumowanie .....	19
7.1. Wnioski .....	19
8. Skrypt .....	21

# 1. Opis problemu

Klasyfikacja win, polegająca na przypisaniu konkretnego trunku do właściwej odmiany winogron na podstawie jego parametrów chemicznych, odgrywa ważną rolę w analizie jakości oraz unikalnych cech wina. Zróżnicowanie cech win pod względem składu chemicznego sprawia, że klasyfikacja win jest trudnym zadaniem.

Dzięki nowoczesnym rozwiązaniom, takim jak sztuczna inteligencja i algorytmy uczenia głębokiego, możliwe jest zautomatyzowanie procesu klasyfikacji win. Problem ten można sprowadzić do zadania wieloklasowej klasyfikacji, w którym model, analizując cechy takie jak zawartość alkoholu, poziom kwasowości czy intensywność barwy, jest w stanie rozpoznać odmianę winogron wykorzystaną do produkcji danego wina.

## 1.1. Istota problemu

Klasyfikacja win na podstawie ich składu chemicznego ma znaczenie zarówno dla producentów, jak i konsumentów. Dla producentów stanowi narzędzie umożliwiające kontrolę jakości, identyfikację charakterystycznych cech produktu oraz optymalizację procesów wytwarzania. Z kolei dla konsumentów wiedza o właściwościach wina pozwala na lepsze dopasowanie do indywidualnych preferencji smakowych.

Automatyzacja tego procesu za pomocą modeli sztucznej inteligencji nie tylko przyspiesza i ułatwia analizę, ale również otwiera nowe możliwości w zakresie badań nad różnorodnością win oraz ich właściwościami.

## 1.2. Technologiczne aspekty rozwiązania

Zastosowanie sztucznej inteligencji, w szczególności algorytmów uczenia maszynowego, stanowi kluczowy element w procesie automatycznej klasyfikacji win. Nowoczesne techniki, takie jak sieci neuronowe czy głębokie uczenie (Deep Learning), umożliwiają modelom analizowanie złożonych wzorców w danych chemicznych, które są trudne do wychwycenia przez tradycyjne metody.

Do implementacji tych sieci najczęściej wykorzystuje się popularne biblioteki, takie jak `PyTorch` i `TensorFlow`, które oferują szeroki zestaw narzędzi do budowy, trenowania i optymalizacji sieci neuronowych. Dodatkowo, `Burn`, biblioteka stworzona w języku `Rust`, staje się coraz bardziej popularną alternatywą do bibliotek w językach wysokopoziomowych, dzięki swojej wydajności i możliwościach języka niskopoziomowego.

## 1.3. Cel projektu

Celem projektu jest opracowanie modelu sztucznej inteligencji, który będzie w stanie automatycznie klasyfikować wina na podstawie ich cech chemicznych. Projekt zakłada stworzenie systemu wykorzystującego algorytmy głębokiego uczenia, który na podstawie danych, precyzyjnie przypisze wino do jednej z określonych odmian. Dążymy do opracowania modelu o wysokiej dokładności, który może zostać wykorzystany zarówno w badaniach naukowych, jak i w przemyśle winiarskim do automatycznej klasyfikacji produktów.

## 2. Opis części praktycznej

### 2.1. Uczenie modelu

Uczenie modelu w kontekście Deep Learningu polega na wykorzystaniu zaawansowanych algorytmów sztucznej inteligencji, szczególnie sieci neuronowych, do rozpoznawania wzorców i zależności w danych. *Deep Learning*, będący poddziedziną uczenia maszynowego, różni się od tradycyjnych metod uczenia maszynowego tym, że automatycznie wykonuje proces ekstrakcji cech z surowych danych, bez potrzeby ręcznego selekcjonowania istotnych atrybutów. Proces ten odbywa się dzięki warstwom sieci neuronowych, które są odpowiedzialne za stopniowe przekształcanie danych wejściowych na reprezentacje coraz bardziej złożone.

Uczenie modelu odbywa się w procesie zwanym treningiem, który polega na minimalizowaniu funkcji błędu. Funkcja ta mierzy, jak bardzo przewidywania modelu różnią się od rzeczywistych etykiet w danych. Podczas treningu sieć neuronowa iteracyjnie dostosowuje swoje parametry (*wagi i biasy*), aby jak najlepiej odwzorować prawdziwe dane wyjściowe. Używa się tutaj algorytmu optymalizacji, najczęściej **spadku gradientu** (*Gradient Descent*), który pozwala na modyfikację wag w taki sposób, by minimalizować błąd predykcji.

W procesie tym kluczową rolę odgrywa funkcja aktywacji, która wprowadza nieliniowość do modelu i pozwala na modelowanie bardziej złożonych zależności. Przykładowe funkcje aktywacji to **ReLU** (*Rectified Linear Unit*), **sigmoid** czy **tanh**, które pomagają w decyzji, czy dane wejściowe mają być uwzględnione w obliczeniach w danej warstwie.

W trakcie treningu model uczy się generalizować, czyli wykrywać ogólne wzorce, które będą skuteczne nie tylko dla danych treningowych, ale także dla nowych, niewidzianych wcześniej przykładów. Aby uniknąć przeuczenia (*overfitting*), stosuje się techniki, takie jak regularizacja, **dropout** czy **early stopping**, które pomagają utrzymać model w równowadze pomiędzy dokładnością a zdolnością do uogólniania.

### 2.2. Nadzorowane uczenie

Uczenie nadzorowane to metoda uczenia maszynowego, w której model jest trenowany na danych wejściowych, które są już przypisane do odpowiednich klas (czyli wyników). W przypadku klasyfikacji win oznacza to, że dla każdego przykładu (wina) znamy już jego kategorię, czyli odmianę winogron,

z której pochodzi. Model wykorzystuje te informacje, aby nauczyć się, jak przypisać nowe, nieznanne dane do właściwej klasy.

W procesie uczenia modelu, sieć neuronowa analizuje cechy chemiczne wina (takie jak zawartość alkoholu, kwasowość, intensywność barwy itp.), a następnie na podstawie tych danych podejmuje decyzję, do której z trzech odmian winogron należy dane wino. Model jest trenowany na wcześniej oznaczonych danych (winach o znanych odmianach), a celem jest minimalizacja błędu klasyfikacji, aby przewidywania modelu były jak najdokładniejsze.

## 2.3. Ocena skuteczności modelu

Testowanie modelu polega na sprawdzeniu jego zdolności do poprawnego przypisywania nowych, niewidzianych wcześniej danych do odpowiednich klas. W tym przypadku model, który został wytrenowany na danych zawierających cechy chemiczne win, jest testowany na osobnym zbiorze testowym, który nie był używany w trakcie treningu. Celem jest ocena, jak dobrze model generalizuje swoje umiejętności na nowych danych, a nie tylko na tych, które były obecne w zestawie treningowym.

Podczas testowania modelu ocenia się jego skuteczność przy pomocy odpowiednich metryk, takich jak **dokładność** (*accuracy*), która wskazuje, jaka część wszystkich przewidywań była poprawna oraz **strata** (*loss*), która określa, jak dobrze model radzi sobie z minimalizacją błędów w klasyfikacji.

### 2.3.1. Dokładność

Dokładność modelu to procent poprawnych przewidywań w stosunku do wszystkich przewidywań. Im wyższa wartość dokładności, tym lepsza jest skuteczność modelu. Na przykład, jeśli model przewidywał poprawnie 90 z 100 próbek, to jego dokładność wynosi 90%. Dokładność jest jedną z najczęściej stosowanych metryk w problemach klasyfikacji, ponieważ jest intuicyjna i łatwa do interpretacji.

$$\text{Accuracy} = \frac{\text{Liczba poprawnych przewidywań}}{\text{Liczba wszystkich próbek}}$$

### 2.3.2. Strata

**Strata** (*loss*) to wartość funkcji błędu, która mierzy, jak bardzo przewidywania modelu różnią się od rzeczywistych etykiet w danych.

Dla problemu klasyfikacji, jedną z najczęściej stosowanych funkcji *loss* jest **cross-entropy loss**, która mierzy różnicę między przewidywaną a rzeczywistą rozkładem prawdopodobieństw dla poszczególnych klas. Im mniejsza wartość *loss*, tym lepsza jest skuteczność modelu, ponieważ oznacza to, że przewidywania modelu są bardziej zbliżone do rzeczywistych etykiet.

### 3. Analiza danych wejściowych

Analiza danych wejściowych to kluczowy etap w każdym projekcie uczenia maszynowego, który pozwala na zrozumienie struktury danych, ich właściwości oraz przygotowanie ich do dalszego przetwarzania i trenowania modelu. W tym przypadku celem analizy jest dokładne zrozumienie cech chemicznych win, które stanowią podstawę do klasyfikacji różnych odmian winogron.

#### 3.1. Opis zbioru danych

Zbiór danych, który jest wykorzystywany w tym projekcie, pochodzi z analizy chemicznej win pochodzących z tego samego regionu we Włoszech, ale z trzech różnych odmian winogron. Zawiera on 13 cech chemicznych, które zostały zmierzone dla każdego z win, takich jak zawartość alkoholu, kwasowość, intensywność barwy oraz inne właściwości, które wpływają na charakterystykę wina. W zbiorze danych znajdują się również etykiety, które informują o rodzaju odmiany winogron, z której pochodzi dane wino.

Dokładnym źródłem danych jest „*PARVUS - An Extendible Package for Data Exploration, Classification and Correlation*”. Instytut Analiz Farmaceutycznych i Żywnościowych oraz Technologii, Genua, Włochy.

#### 3.2. Struktura zbioru danych

Każdy wiersz w zbiorze danych reprezentuje jedno wino, a 13 kolumn odpowiada różnym cechom chemicznym, które zostały zmierzone w próbce. Wartości w tych kolumnach są liczbowe, co oznacza, że dane są już w formie numerycznej, gotowe do dalszego przetwarzania. Również warto wspomnieć, że nie ma brakujących danych w zbiorze, co ułatwia analizę.

Zbiór danych podzielony jest na 178 próbek, a liczby próbek dla poszczególnych odmian winogron to:

- Klasa 1: 59 próbek - (33.1%)
- Klasa 2: 71 próbek - (39.9%)
- Klasa 3: 48 próbek - (27.0%)

Atrybuty chemiczne, które zostały zmierzone dla każdego wina, to:

1. **Alkohol** – Zawartość alkoholu w winie, wyrażona jako procent objętości alkoholu w stosunku do całkowitej objętości płynu. Jest to jeden z kluczowych wskaźników wpływających na smak i charakterystykę wina.
2. **Kwas jabłkowy** – Zawartość kwasu jabłkowego w winie. Jest to naturalny kwas organiczny, który wpływa na smak wina, nadając mu kwaskowość. Wina o wyższej zawartości tego kwasu mogą być bardziej kwasowe i orzeźwiające.
3. **Popiół** – Zawartość popiołu w winie, który jest pozostałością po spaleniu organicznych składników w winie. Zawartość popiołu może mieć wpływ na mineralność wina.
4. **Zasadowość popiołu** – Określa poziom zasadowości popiołu w winie. Zasadowość wpływa na pH wina i może oddziaływać na smak, zwłaszcza w kontekście równowagi kwasowości i słodczy.

5. **Magnez** – Zawartość magnezu w winie. Magnez jest jednym z minerałów, który wpływa na smak wina, a jego obecność może wpływać na ogólną jakość i strukturę wina.
6. **Całkowita zawartość fenoli** – Fenole są naturalnymi związkami organicznymi występującymi w winie, które odpowiadają za smak, zapach oraz właściwości zdrowotne wina. Ich zawartość może wpływać na goryczkę oraz astringencję wina.
7. **Flawanoidy** – Jest to grupa fenoli, która wpływa na smak, zapach i kolor wina. Flawanoidy są również antyoksydantami, co ma znaczenie dla trwałości wina.
8. **Fenole nieflawanoidowe** – Inna grupa fenoli, która nie należy do flawanoidów, ale również wpływa na smak i zapach wina, nadając mu specyficzne cechy organoleptyczne.
9. **Proantocyjaniny** – Związki odpowiedzialne za kolor wina, zwłaszcza czerwonych win. Są to silne antyoksydanty, które również wpływają na strukturę smaku i astringencję.
10. **Intensywność koloru** – Określa głębokość koloru wina, co jest ważnym atrybutem w przypadku win czerwonych i białych. Intensywność koloru może wskazywać na dojrzałość wina oraz jego skład chemiczny.
11. **Odcień** – Dotyczy barwy wina, który może się różnić w zależności od odmiany winogron, procesu fermentacji i starzenia wina. Odcień jest kluczowym elementem oceny jakości wina.
12. **OD280/OD315 rozcieńczonych win** – Stosunek absorbancji przy długości fali 280 nm do 315 nm, który jest wykorzystywany do oceny jakości i zawartości substancji organicznych w winie, takich jak fenole.
13. **Prolina** – Aminokwas występujący w winie, który wpływa na strukturę wina oraz jego stabilność. Zawartość proliny może mieć wpływ na właściwości smakowe i chemiczne wina.

## 4. Przygotowanie danych do nauki

Odpowiednie przygotowanie danych ma na celu zapewnienie, że model będzie w stanie uczyć się skutecznie i osiągać jak najlepsze wyniki. W tym etapie dokonuje się kilku istotnych operacji, takich jak czyszczenie danych, normalizacja, dodanie brakujących wartości, podział na zbiory treningowe i testowe, a także inne techniki, które pomagają w dostosowaniu danych do wymagań modelu.

Poniżej przedstawię kroki, które ja podjąłem w celu przygotowania danych do trenowania modelu klasyfikacji win.

### 4.1. Czyszczenie danych

Pierwszym krokiem w przygotowaniu danych jest ich czyszczenie, czyli usunięcie zbędnych informacji, brakujących wartości czy duplikatów. W przypadku zbioru danych, który został wykorzystany w tym projekcie, nie było potrzeby usuwania żadnych próbek. Każdy atrybut może potencjalnie wpłynąć na klasyfikację wina, dlatego nie ma potrzeby eliminacji.

Jedynym ważnym aspektem była zmiana etykiet klas, aby numeracja zaczynała się od 0, co jest wymagane przez bibliotekę `Burn`. Wcześniej etykiety klas były oznaczone jako 1, 2 i 3, co zostało zmienione na 0, 1 i 2.

## 4.2. Normalizacja danych

Normalizacja danych jest ważnym krokiem w przypadku modeli, które wykorzystują algorytmy uczenia maszynowego, ponieważ pozwala na ujednolicenie skali wartości atrybutów. W przypadku zbioru danych z cechami chemicznymi win, wartości poszczególnych atrybutów różnią się znacząco, co może wpłynąć na proces uczenia modelu.

Np. zawartość alkoholu w winie mieści się w zakresie od 11.0% do 14.8%, podczas gdy zawartość kwasu jabłkowego waha się od 1.74 do 4.23. Aby uniknąć problemów związanych z różnicą w skali wartości, zastosowałem normalizację.

W procesie normalizacji dane są przeskalowane w taki sposób, aby mieściły się w określonym zakresie. W moim przypadku od  $-1$  do  $1$ .

$$x_{\text{norm}} = \frac{2 \cdot (x - x_{\min})}{x_{\max} - x_{\min}} - 1$$

*Wzór 1: Normalizacji danych do zakresu  $[-1, 1]$*

```
1  for (i, row) in x.iter().enumerate() {
2      for (j, &value) in row.iter().enumerate() {
3          if max[j] - min[j] == 0.0 {
4              x_norm[i][j] = 0.0; // Avoid division by zero
5          } else {
6              // Normalize to range [-1, 1]
7              x_norm[i][j] = -1.0 + 2.0 * (value - min[j]) / (max[j] - min[j]);
8          }
9      }
10 }
```

*Program 1: Normalizacja danych*



## 4.3. Podział na zbiory treningowe i testowe

Podział danych na zbiory treningowe i testowe jest kluczowym elementem w procesie uczenia maszynowego, ponieważ pozwala na ocenę skuteczności modelu na nowych, niewidzianych wcześniej danych. W moim przypadku zdecydowałem się na podział danych w stosunku 80% do 20%, gdzie 80% danych zostało wykorzystane do treningu, a 20% do testowania. W późniejszych etapach zmieniałem ten podział na 70% do 30% w celach eksperymentów.

Podział danych na zbiory treningowe i testowe pozwala na ocenę skuteczności modelu na nowych, niewidzianych wcześniej danych. W ten sposób można sprawdzić, jak dobrze model generalizuje swoje umiejętności na nowych danych, a nie tylko na tych, które były obecne w zestawie treningowym.

W tym celu napisałem funkcję `split_data`, która dokonuje losowego podziału danych na zbiory treningowe i testowe. Każda klasa jest losowana osobno, aby zachować równowagę między klasami w obu zbiorach.

```
1 fn split_data(Rust
2     x: Vec<Vec<f32>>,
3     y_t: Vec<i32>,
4 ) → (Vec<Vec<f32>>, Vec<i32>, Vec<Vec<f32>>, Vec<i32>) {
5     let class_counts = get_class_count(&y_t);
6     let mut x_train = Vec::new();
7     let mut y_t_train = Vec::new();
8     let mut x_test = Vec::new();
9     let mut y_t_test = Vec::new();
10
11     let wines = x.iter().zip(y_t.iter()).collect::<Vec<_>>();
12
13     for (class, count) in class_counts.iter() {
14         let num_train = (count * 85) / 100;
15
16         let mut class_wines = wines.iter().filter(|(_, &y)| y == *class).collect::<Vec<_>>();
17         class_wines.shuffle(&mut thread_rng());
18
19         let (test, train) = class_wines.split_at(num_train as usize);
20
21         for (x, y) in train.into_iter() {
22             x_train.push(x.clone().clone());
23             y_t_train.push(y.clone().clone());
24         }
25
26         for (x, y) in test.iter() {
27             x_test.push(x.clone().clone());
28             y_t_test.push(y.clone().clone());
29         }
30     }
31
32     (x_train, y_t_train, x_test, y_t_test)
33 }
```

Program 2: Podział danych na zbiory treningowe i testowe

## 4.4. Zapis danych do plików

Tak przygotowane dane zostały zapisane do plików w formacie PKL. Do zserializowania danych wykorzystałem bibliotekę `serde-pickle` z pakietu `serde`, która pozwala na serializację i deserializację danych w formacie PKL. Dane zostały zapisane w postaci czterech plików: `x_train.pkl`, `y_train.pkl`, `x_test.pkl` i `y_test.pkl`.

```
1  /// Save `x` and `y_t` to disk in pickle format. Rust
2  fn dump_to_pkl(x: Vec<Vec<f32>>, y_t: Vec<i32>, prefix: &str) {
3      // Create BTreeMaps to serialize
4      let mut x_map = BTreeMap::new();
5      let mut y_map = BTreeMap::new();
6      x_map.insert(String::from("x"), x);
7      y_map.insert(String::from("y_t"), y_t);
8
9      // Serialize to pickle format
10     let x_serialized = serde_pickle::to_vec(&x_map, Default::default()).unwrap();
11     let y_serialized = serde_pickle::to_vec(&y_map, Default::default()).unwrap();
12
13     // Save to disk
14     std::fs::write(format!("./data/x-{}.pkl", prefix), &x_serialized).unwrap();
15     std::fs::write(format!("./data/y_t-{}.pkl", prefix), &y_serialized).unwrap();
16 }
```

Program 3: Funkcja zapisująca dane do plików PKL

## 4.5. Podsumowanie

Przygotowanie danych do trenowania modelu klasyfikacji win wymagało kilku istotnych kroków, takich jak czyszczenie, normalizacja, podział na zbiory treningowe i testowe oraz zapis do plików. Dzięki tym operacjom dane są gotowe do dalszego przetwarzania i trenowania modelu.

y_t	alcohol	malic acid	ash	alcalinity of ash	magnesium	total phenols	flavanoids	nonflavanoid phenols	proanthocyanins	color intensity	hue	od280 od315 of diluted wines	proline
0	0.44210553	-0.541592	0.41176474	-0.32989687	-0.021739125	0.39310336	0.033755382	-0.01886883	-0.19873816	-0.14334464	0.056910515	0.21611726	0.5649873
0	0.39473712	-0.56917	0.069518566	-0.3195876	-0.26086956	-0.0068965554	-0.008438706	0.09433949	-0.01577276	-0.5651399	0.21951222	0.1721611	0.015691876
0	0.4736842	-0.64031625	0.3262031	-0.3195876	-0.47826087	0.013793111	0.11814356	-0.6603774	0.18611991	-0.26279873	0.23577237	0.53846145	0.40798855
0	0.76315796	-0.55335975	0.09090924	-0.85567015	-0.3043478	0.59999999	0.39240515	-0.39622647	0.60883284	0.061433434	0.17073178	0.2673993	0.010271
0	-0.04210496	-0.66007986	0.24044171	-0.25773203	-0.45652175	0.034482718	-0.14345992	-0.509454	-0.33753937	-0.5477816	-0.008130074	0.72893786	0.051355243
0	0.0736841	-0.69968475	-0.2085563	-0.4948454	-0.39138437	-0.020689607	-0.02953571	-0.4339623	-0.39432174	-0.5870307	0.13821125	0.040293097	0.058487892
0	0.00000023841858	0.20948601	0.3796792	-0.17525774	-0.3043478	-0.013793111	-0.12658215	-0.5471698	-0.009463668	-0.45051193	-0.105691135	0.64835167	-0.29814553
0	0.75789475	-0.5217391	0.21925128	-0.36082482	-0.065217376	0.97931015	0.32911408	-0.5849057	0.116719365	0.11262798	-0.3821138	0.5970696	0.7146933
0	0.72105277	-0.5335969	0.4545455	-0.030927837	0.0869565	0.25517237	0.18143475	-0.24528301	-0.01577276	-0.10400957	-0.040650368	0.01098907	0.42938662
1	-0.5105262	-0.06166006	0.0053474903	0.07216489	-0.32608694	0.05517247	-0.24050629	-1	-0.21766555	-0.67064846	-0.17073172	0.3626374	-0.1326676
1	-0.3578946	-0.6086957	-0.18716592	-0.13402063	-0.7826087	-0.5379311	-0.28691983	-0.09433967	-0.23028386	-0.63822526	-0.15447158	0.39194143	-0.66904426
1	-0.08421022	0.063241005	-0.3368984	-0.443299	-0.7826087	-0.5517242	-0.0160338	0.13207543	-0.73501575	-0.63822526	-0.6422764	-0.37728947	-0.8659059

Rysunek 1: Tabela z częścią danych przygotowanych do trenowania modelu

## 5. Trenowanie modelu

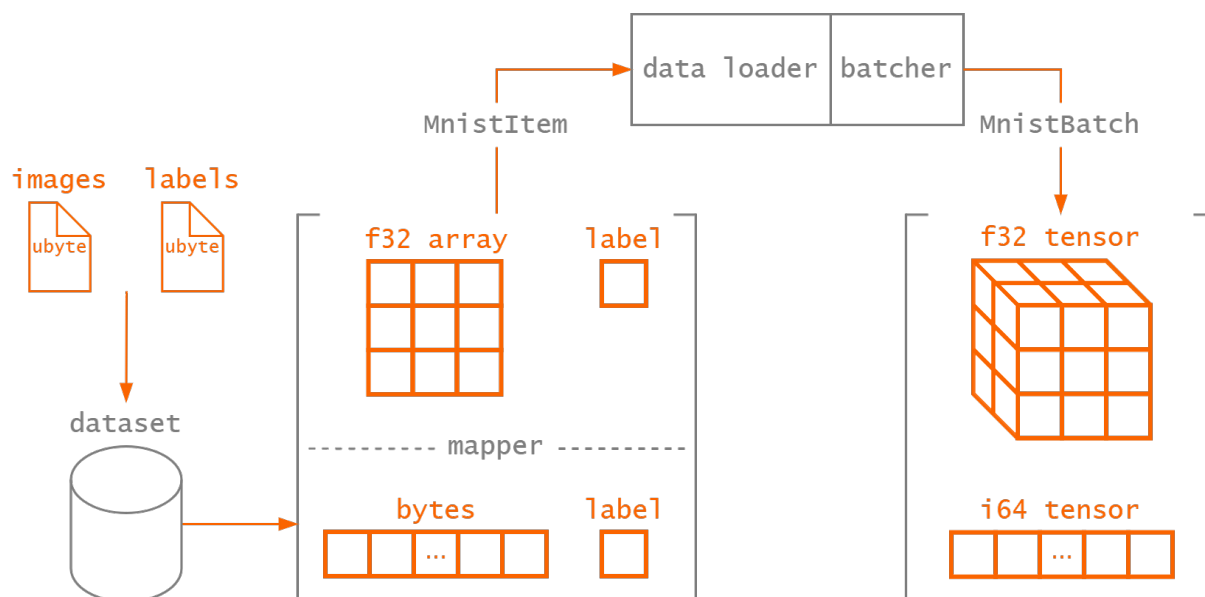
Po przygotowaniu danych przyszedł czas na trenowanie modelu. W tym projekcie zdecydowałem się na wykorzystanie biblioteki `Burn` do implementacji modelu, jak i jego trenowania.

Przed samym przystąpieniem do trenowania modelu, potrzebne jest zdefiniowanie architektury sieci neuronowej, przygotowanie danych treningowych i testowych oraz wybór odpowiednich hiperparametrów, takich jak współczynnik uczenia, liczba epok czy rozmiar wsadu (*batch size*).

### 5.1. Przygotowanie danych

Trenowanie modelu w bibliotece `Burn` wymaga tzw. *Batcherów* (plik `batcher.rs` w części skryptowej), które są odpowiedzialne za dostarczanie danych do modelu w postaci wsadów (*batches*). *Batcher* przyjmuje strukturę *Dataset* (plik `dataset.rs`, w części skryptowej) jako dane wejściowe i dzieli je na wsady o określonym rozmiarze.

W moim przypadku, struktura `WineDataset` przechowuje w pamięci systemowej dane treningowe i testowe, które zostały wcześniej przygotowane i zapisane do plików PKL. Następnie, `WineBatcher` dzieli te dane na wsady `WineBatch`, które są przekazywane do modelu podczas treningu.



Rysunek 2: Schemat przepływu danych w procesie trenowania modelu

#### 5.1.1. Implementacja tensorów

Ważnym elementem w trenowaniu modelu jest implementacja tensorów, które są podstawowymi strukturami danych w bibliotece `Burn`. Tensor reprezentuje wielowymiarowy wektor lub macierz, który jest przechowywany w pamięci urządzenia. W przypadku mojego programu tensory tworzone są w funkcji `batch` w strukturze `WineBatcher`. Przechowują one dane wejściowe i wyjściowe dla modelu, które są przekazywane do sieci neuronowej.

Pierwsze tensor jest tworzony dla każdego wina z danych treningowych, a drugi tensor przechowuje etykietę klasy dla tego wina. Następnie, te tensory są łączone w jeden tensor dla danych wejściowych i wyjściowych, który jest przekazywany bezpośrednio do modelu.

```
1  impl<B: Backend> Batcher<WineItem, WineBatch<B>> for WineBatcher<B> {  Rust
2  fn batch(&self, data: Vec<WineItem>) -> WineBatch<B> {
3      let x = data
4          .iter()
5          .map(|wine| {
6              Tensor::<B, 2>::from_data(
7                  [[
8                      wine.alcohol,
9                      wine.malic_acid,
10                     wine.ash,
11                     wine.alcalinity_of_ash,
12                     wine.magnesium,
13                     wine.total_phenols,
14                     wine.flavanoids,
15                     wine.nonflavanoid_phenols,
16                     wine.proanthocyanins,
17                     wine.color_intensity,
18                     wine.hue,
19                     wine.od280_od315_of_diluted_wines,
20                     wine.proline,
21                 ]],
22                 &self.device,
23             )
24         })
25         .collect::<Vec<_>>();
26
27     let y = data
28         .iter()
29         .map(|wine| Tensor::<B, 1, Int>::from_data([wine.class], &self.device))
30         .collect::<Vec<_>>();
31
32     let x = Tensor::<B, 2>::cat(x, 0).to_device(&self.device);
33     let y = Tensor::<B, 1, Int>::cat(y, 0).to_device(&self.device);
34
35     WineBatch { x, y }
36 }
37 }
```

Program 4: Implementacji funkcji `batch`, której wymaga struktura `WineBatcher`

## 5.2. Definicja modelu

Model sieci neuronowej jest zdefiniowany w strukturze `WineModel` (plik `model.rs` w części skryptowej). Składa się z trzech warstw sieci neuronowej: jednej warstwy wejściowej, jednej warstwy ukrytej i jednej warstwy wyjściowej.

Warstwa wejściowa ma 13 neuronów, które odpowiadają 13 cechom chemicznym wina. Warstwa ukryta ma liczbę neuronów, która jest określona przez hiperparametr `hidden_size`. Na początku eksperymentów przyjąłem wartość `hidden_size = 64`. Warstwa wyjściowa ma 3 neurony, które odpowiadają trzem klasom winogron.

Model wykorzystuje funkcję aktywacji **ReLU** (*Rectified Linear Unit*) w warstwie ukrytej i wejściowej oraz funkcję regulacji **Dropout** w celu zapobiegania przeuczeniu. Funkcja **Dropout** losowo wyłącza część neuronów w warstwie ukrytej podczas treningu, co pomaga w regularyzacji modelu i poprawia jego zdolność do generalizacji.

```
1  impl<B: Backend> WineModel<B> {
2      pub fn forward(&self, x: Tensor<B, 2>) → Tensor<B, 2> {
3          let x = self.input_layer.forward(x);
4          let x = self.activation.forward(x);
5          let x = self.dropout.forward(x);
6
7          let x = self.hidden_layer.forward(x);
8          let x = self.activation.forward(x);
9          let x = self.dropout.forward(x);
10
11         self.output_layer.forward(x)
12     }
13 }
```



Program 5: Implementacji funkcji `forward`, która przekazuje dane przez warstwy sieci neuronowej

## 5.3. Trenowanie modelu

Trenowanie modelu polega na iteracyjnym dostosowywaniu wag i biasów sieci neuronowej w celu minimalizacji funkcji błędu. W tym projekcie wykorzystałem algorytm optymalizacji **Adam**, który jest jednym z najczęściej stosowanych algorytmów optymalizacji w uczeniu maszynowym. **Adam** łączy zalety algorytmów **RMSprop** i **Adagrad**, co pozwala na skuteczne aktualizowanie wag sieci neuronowej.

W trakcie trenowania modelu obliczana jest wartość funkcji straty (*loss*), która mierzy, jak bardzo przewidywania modelu różnią się od rzeczywistych etykiet w danych. Następnie, wartość straty jest wykorzystywana do aktualizacji wag sieci neuronowej w procesie optymalizacji.

**Burn** dostarcza wiele wbudowanych metryk, które można wykorzystać do oceny skuteczności modelu podczas treningu. W moim przypadku, wykorzystałem metryki **dokładność** (*accuracy*) i **strata** (*loss*), które są kluczowe w problemach klasyfikacji.

**Burn** również dostarcza bardzo przejrzysty interfejs pokazujący przebieg treningu, w tym wartości metryk, czas treningu, aktualizacje wag oraz inne informacje, które są przydatne podczas analizy wyników.

```
1 pub fn train<B: AutodiffBackend>(artifact_dir: &str, config: TrainingConfig,
2     device: B::Device) {
3     create_artifact_dir(artifact_dir);
4     config
5     .save(format!("{artifact_dir}/config.json"))
6     .expect("Config should be saved successfully");
7     B::seed(config.seed);
8
9     let train_batch = WineBatcher::<B>::new(device.clone());
10    let test_batch = WineBatcher::<B::InnerBackend>::new(device.clone());
11
12    let data_loader_train = DataLoaderBuilder::new(train_batch)
13        .batch_size(config.batch_size)
14        .shuffle(config.seed)
15        .num_workers(config.num_workers)
16        .build(WineDataset::train());
17
18    let data_loader_test = DataLoaderBuilder::new(test_batch)
19        .batch_size(config.batch_size)
20        .shuffle(config.seed)
21        .num_workers(config.num_workers)
22        .build(WineDataset::test());
23
24    let learner = LearnerBuilder::new(artifact_dir)
25        .metric_train_numeric(AccuracyMetric::new())
26        .metric_valid_numeric(AccuracyMetric::new())
27        .metric_train_numeric(LossMetric::new())
28        .metric_valid_numeric(LossMetric::new())
29        .with_file_checkpoint(CompactRecorder::new())
30        .devices(vec![device.clone()])
```

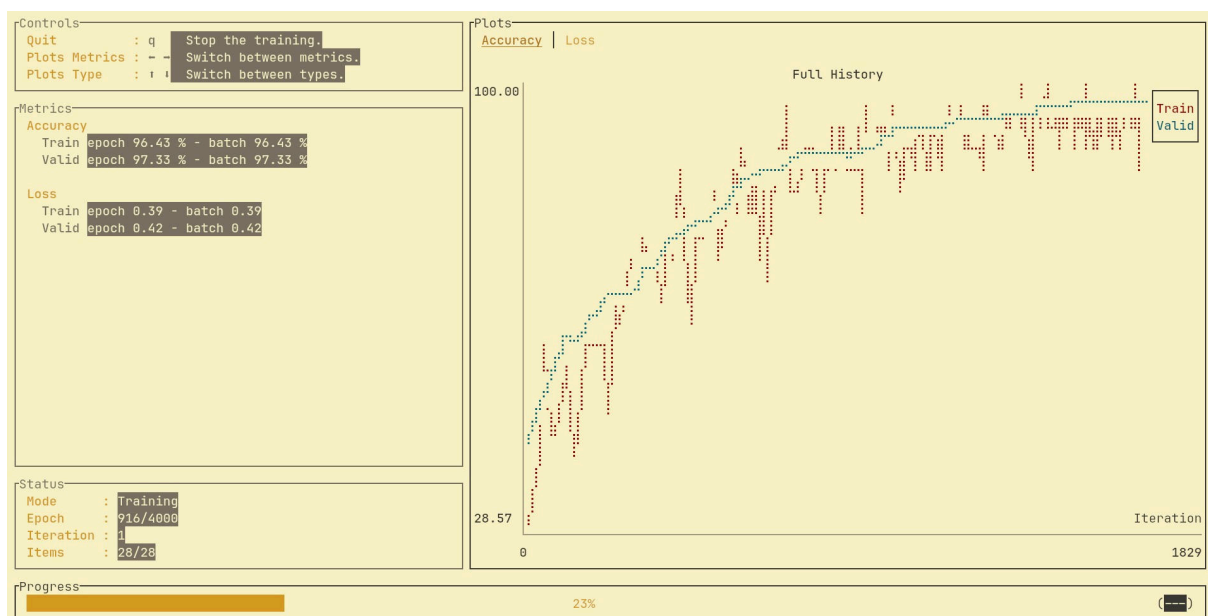


```

31     .num_epochs(config.num_epochs)
32     .summary()
33     .build(
34         config.model.init::<B>(&device),
35         config.optimizer.init(),
36         config.learning_rate,
37     );
38
39     let model_trained = learner.fit(dataloader_train, dataloader_test);
40
41     model_trained
42         .save_file(format!("{artifact_dir}/model"), &CompactRecorder::new())
43         .expect("Trained model should be saved successfully");
44 }

```

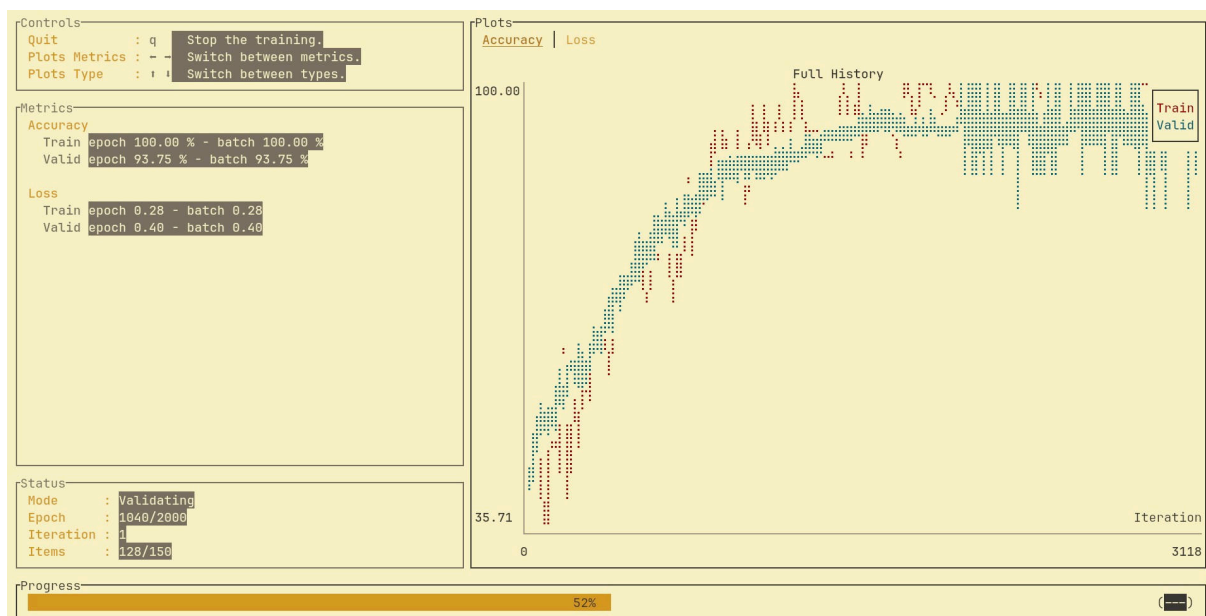
Program 6: Funkcja `train`, która trenuje model sieci neuronowej



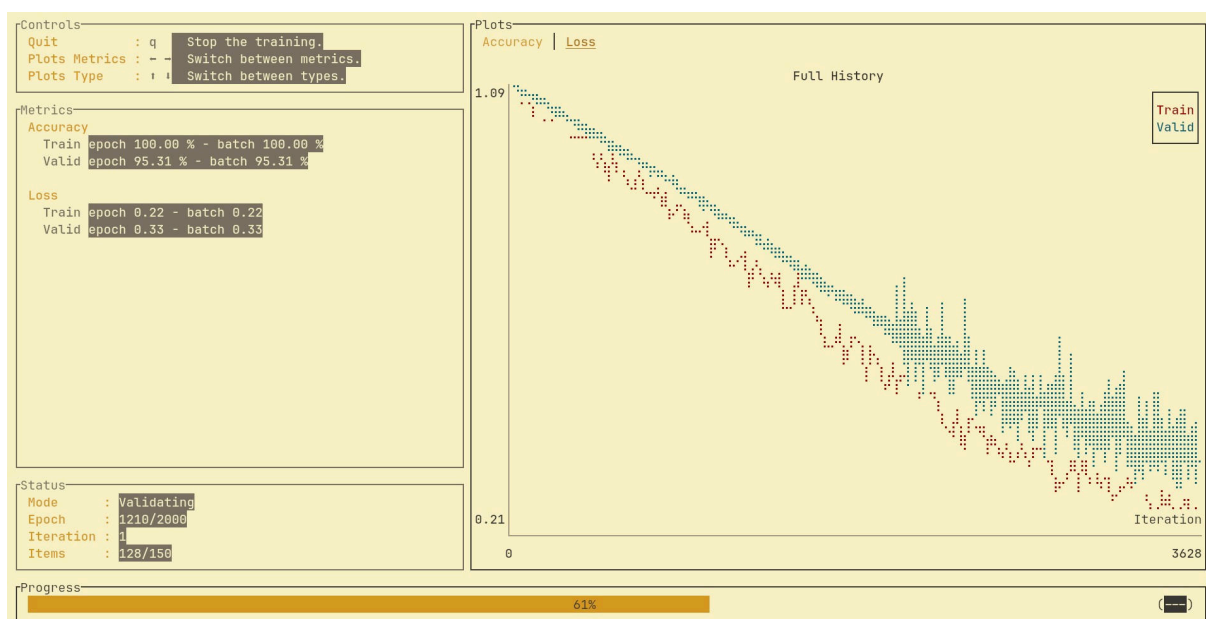
Rysunek 3: Interfejs `Burn` podczas treningu modelu

### 5.3.1. Wyniki trenowania

Po zakończeniu trenowania modelu można ocenić jego skuteczność na zbiorze testowym. Początkowo wykorzystałem hiperparametry `hidden_size = 64`, `num_epochs = 2000`, `batch_size = 128` i `learning_rate = 1.0e-3`. Te parametry pozwoliły mi uzyskać dokładność na poziomie około 100% na zbiorze treningowym, a 94% na zbiorze testowym. Wyniki te były obiecujące, ale postanowiłem przeprowadzić serię eksperymentów, aby znaleźć optymalne wartości hiperparametrów.



Rysunek 4: Wykres dokładności modelu dla hiperparametrów  $hidden\_size = 64$ ,  $num\_epochs = 2000$ ,  $batch\_size = 128$  i  $learning\_rate = 1e-3$



Rysunek 5: Wykres straty modelu dla hiperparametrów  $hidden\_size = 64$ ,  $num\_epochs = 2000$ ,  $batch\_size = 128$  i  $learning\_rate = 1e-3$

Split	Metric	Min.	Epoch	Max.	Epoch
Train	Loss	0.063	1950	1.112	2
Train	Accuracy	21.429	62	100.000	2000
Valid	Loss	0.175	1958	1.095	1
Valid	Accuracy	41.333	1	94.000	1001

Rysunek 6: Tabela wyników modelu dla hiperparametrów  $hidden\_size = 64$ ,  $num\_epochs = 2000$ ,  $batch\_size = 128$  i  $learning\_rate = 1e-3$



## 6. Eksperymenty

W celu poprawy skuteczności modelu przeprowadziłem serię eksperymentów, w których zmieniałem hiperparametry modelu i nie tylko. Zmiany te miały na celu znalezienie optymalnych wartości, które pozwolą na uzyskanie jak najlepszych wyników.

### 6.1. Generowanie danych

Pierwszym eksperymentem było zbadanie kilku różnych prób danych treningowych i testowych, aby sprawdzić, jak zmiana podziału danych wpłynie na skuteczność modelu. W trakcie eksperymentów zmieniałem stosunek danych treningowych do testowych, aby sprawdzić, czy zmiana proporcji ma wpływ na wyniki modelu.

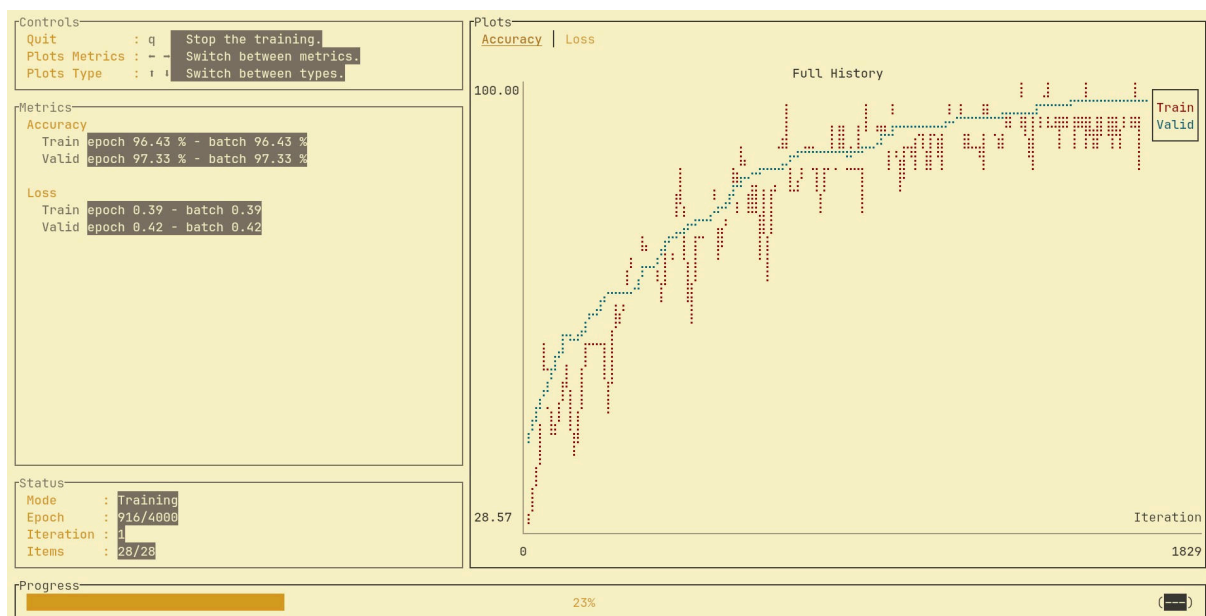
Ostatecznie, najlepszym stosunkiem okazał się podział 85% danych treningowych do 15% danych testowych. Ten podział pozwolił mi uzyskać najlepsze wyniki.

Zauważyłem też, że wyniki potrafiły się różnić bez zmiany stosunku podziału, a po samym wygenerowaniu ponownie danych. Wynika to z losowości podziału danych na zbiory treningowe i testowe, co może wpływać na wyniki modelu.

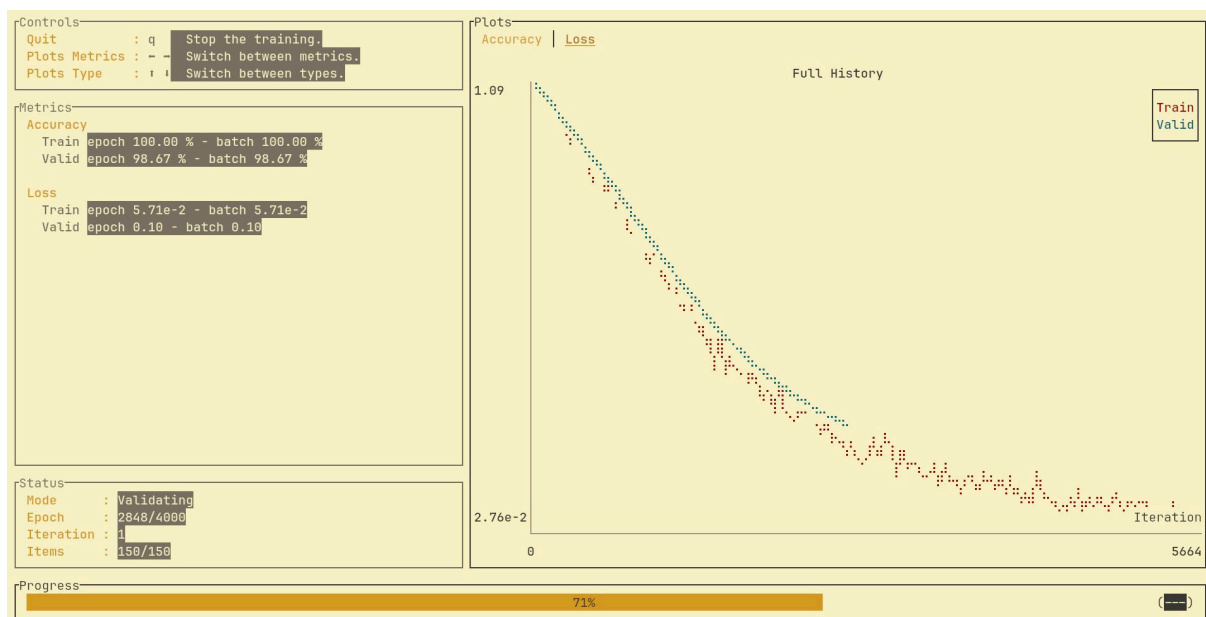
### 6.2. Zmiana hiperparametrów

Pierwszym krokiem w eksperymentach było zmienienie hiperparametrów modelu, takich jak `hidden_size`, `num_epochs`, `batch_size` i `learning_rate`. W trakcie eksperymentów testowałem różne wartości tych parametrów, aby znaleźć optymalne wartości, które pozwolą na uzyskanie jak najlepszych wyników.

Odkryłem, że najlepszymi wartościami hiperparametrów są `hidden_size = 64`, `num_epochs = 4000`, `batch_size = 256` i `learning_rate = 1.0e-4`. Te wartości pozwoliły mi uzyskać dokładność na poziomie około 100% na zbiorze treningowym, a 98.667% na zbiorze testowym. Strata modelu była również na niskim poziomie - około  $8.802e-3$  dla zbioru treningowego i 0.081 dla zbioru testowego.



Rysunek 7: Wykres dokładności modelu dla hiperparametrów  $hidden\_size = 64$ ,  $num\_epochs = 4000$ ,  $batch\_size = 256$  i  $learning\_rate = 1e-4$



Rysunek 8: Wykres straty modelu dla hiperparametrów  $hidden\_size = 64$ ,  $num\_epochs = 4000$ ,  $batch\_size = 256$  i  $learning\_rate = 1e-4$

Split	Metric	Min.	Epoch	Max.	Epoch
Train	Accuracy	17.857	18	100.000	4000
Train	Loss	8.802e-3	3808	1.120	2
Valid	Accuracy	41.333	1	98.667	3171
Valid	Loss	0.081	4000	1.090	1

Rysunek 9: Tabela wyników modelu dla hiperparametrów  $hidden\_size = 64$ ,  $num\_epochs = 4000$ ,  $batch\_size = 256$  i  $learning\_rate = 1e-4$

## 6.3. Zmiana architektury modelu

Kolejnym krokiem w eksperymentach było zmienienie architektury modelu, tak aby sprawdzić, czy zmiana liczby neuronów w warstwie ukrytej wpłynie na skuteczność modelu. W trakcie eksperymentów testowałem różne wartości parametru `hidden_size`, aby znaleźć optymalną wartość, która pozwoli na uzyskanie jak najlepszych wyników.

Dodanie kolejnego neuronu w warstwie ukrytej nie zmieniło znacząco wyników modelu. Ostatecznie, najlepszymi wartościami hiperparametrów okazały się `hidden_size = 64`, `num_epochs = 4000`, `batch_size = 256` i `learning_rate = 1.0e-4`.

Następnie spróbowałem usunąć warstwę ukrytą z modelu, aby sprawdzić, czy prostsza architektura sieci neuronowej będzie skuteczniejsza. Okazało się, że model bez warstwy ukrytej uzyskał gorsze wyniki niż model z warstwą ukrytą, ale różnica nie była znacząca. Ostatecznie, zdecydowałem się na model z warstwą ukrytą, ponieważ uzyskał najlepsze wyniki.

## 7. Podsumowanie

W niniejszej pracy omówiono proces tworzenia i testowania modelu klasyfikacyjnego przy użyciu danych chemicznych dotyczących różnych rodzajów winogron. Dzięki zastosowaniu nowoczesnych technik sztucznej inteligencji, takich jak głębokie uczenie, możliwe było zbudowanie modelu, który skutecznie klasyfikuje wina na podstawie ich atrybutów chemicznych. W ramach przygotowania danych do nauki omówiono m.in. techniki normalizacji, które miały na celu zapewnienie, że cechy wejściowe mają jednolitą skalę i nie wpływają negatywnie na proces uczenia modelu. Testowanie modelu uwzględniało m.in. miarę loss oraz dokładność, które pozwoliły na ocenę skuteczności modelu.

W pracy uwzględniono także kluczowe aspekty technologiczne związane z używaną biblioteką - `Burn`. Przygotowanie danych i ich odpowiednia obróbka są niezbędne, by model mógł działać skutecznie, a wyniki testów wskazują na pozytywne efekty zastosowanych rozwiązań w kontekście klasyfikacji win. Dzięki tym technologiom możliwe jest uzyskanie dokładnych prognoz dotyczących typu wina, co może mieć zastosowanie w różnych dziedzinach, od przemysłu winiarskiego po zastosowania badawcze w naukach chemicznych.

### 7.1. Wnioski

1. **Efektywność sztucznej inteligencji w klasyfikacji danych chemicznych** – Zastosowanie głębokiego uczenia do klasyfikacji win na podstawie ich atrybutów chemicznych okazało się skuteczne. Model uzyskał wysoką dokładność, co sugeruje, że takie podejście może być użyteczne w analizach przemysłowych oraz badaniach naukowych.
2. **Znaczenie przygotowania danych** – Dobrze przeprowadzone przygotowanie danych, w tym normalizacja i odpowiedni podział na zbiory treningowe oraz testowe, miało kluczowy wpływ na wydajność modelu. W szczególności normalizacja danych pozwoliła na poprawienie stabilności modelu i zapewnienie, że wszystkie cechy miały podobną wagę.

3. **Potencjał do zastosowań praktycznych** – Modele klasyfikacyjne oparte na głębokim uczeniu mogą mieć szerokie zastosowanie w przemyśle winiarskim, gdzie dokładne określenie rodzaju wina na podstawie jego właściwości chemicznych może pomóc w procesach produkcji, kontroli jakości i marketingu.

## 8. Skrypt

```
1 use cli_table::*;
2 use si_project::dataset::WineItem;
3 use std::collections::BTreeMap;
4 use std::default::Default;
5 use std::fs::read_to_string;
6
7 fn main() {
8     let data = load_data_file();
9     let wines = parse_data(data);
10
11     // 1. Extract `x` and `y_t`
12     let (x, y_t) = extract_x_and_y_t(&wines);
13
14     // 2. Normalize `x` → `x_norm`
15     let x_norm = normalize_x(&x);
16
17     // Sorting by `y_t` is unnecessary for this dataset. Data is already sorted by class.
18
19     // 3. Split data into training and testing sets
20     let (x_train, y_t_train, x_test, y_t_test) = split_data(x_norm, y_t);
21
22     // 3. Display data in a table
23     println!("Data test:");
24     table_data(&x_train, &y_t_train);
25     println!("\n\nData train:");
26     table_data(&x_test, &y_t_test);
27
28     // 4. Save everything to an PKL file
29     dump_to_pkl(x_train, y_t_train, "train");
30     dump_to_pkl(x_test, y_t_test, "test");
31
32     println!("Data processing and saving completed.");
33 }
34
35 /// Load data file into a `String`.
36 fn load_data_file() → String {
37     read_to_string("./data/wine.data").unwrap()
38 }
39
40 /// Parse raw data into a Vector of `Wine` structs.
41 fn parse_data(data: String) → Vec<WineItem> {
42     data.lines()
43         .map(|line| {
44             let mut wine_data = line.split(',').map(|s| s.parse::<f32>().unwrap());
45             WineItem {
46                 class: wine_data.next().unwrap() as i32,
47                 alcohol: wine_data.next().unwrap(),
48                 malic_acid: wine_data.next().unwrap(),
49                 ash: wine_data.next().unwrap(),
50                 alcalinity_of_ash: wine_data.next().unwrap(),
51                 magnesium: wine_data.next().unwrap(),
52                 total_phenols: wine_data.next().unwrap(),
53                 flavanoids: wine_data.next().unwrap(),
54                 nonflavanoid_phenols: wine_data.next().unwrap(),
55                 proanthocyanins: wine_data.next().unwrap(),
56                 color_intensity: wine_data.next().unwrap(),
57                 hue: wine_data.next().unwrap(),
58                 od280_od315_of_diluted_wines: wine_data.next().unwrap(),
59                 proline: wine_data.next().unwrap(),
60             }
61         })
62 }
```

```

61     })
62     .collect()
63 }
64
65 /// Extract `x` (Wine attributes 1-13) and `y_t` (class attribute).
66 fn extract_x_and_y_t(wines: &[WineItem]) → (Vec<Vec<f32>>, Vec<i32>) {
67     let num_records = wines.len();
68     let mut x = vec![vec![0.0; 13]; num_records];
69     let mut y_t = vec![0; num_records];
70
71     for (i, wine) in wines.iter().enumerate() {
72         x[i][0] = wine.alcohol;
73         x[i][1] = wine.malic_acid;
74         x[i][2] = wine.ash;
75         x[i][3] = wine.alcalinity_of_ash;
76         x[i][4] = wine.magnesium;
77         x[i][5] = wine.total_phenols;
78         x[i][6] = wine.flavanoids;
79         x[i][7] = wine.nonflavanoid_phenols;
80         x[i][8] = wine.proanthocyanins;
81         x[i][9] = wine.color_intensity;
82         x[i][10] = wine.hue;
83         x[i][11] = wine.od280_od315_of_diluted_wines;
84         x[i][12] = wine.proline;
85
86         y_t[i] = wine.class - 1;
87     }
88
89     (x, y_t)
90 }
91
92 /// Normalize `x` (array of wine attributes 1-13).
93 fn normalize_x(x: &Vec<Vec<f32>>) → Vec<Vec<f32>> {
94     let num_columns = x[0].len();
95
96     let mut min = vec![f32::MAX; num_columns];
97     let mut max = vec![f32::MIN; num_columns];
98
99     for row in x.iter() {
100         for (j, &value) in row.iter().enumerate() {
101             if value < min[j] {
102                 min[j] = value;
103             }
104             if value > max[j] {
105                 max[j] = value;
106             }
107         }
108     }
109
110     let mut x_norm = x.clone();
111     for (i, row) in x.iter().enumerate() {
112         for (j, &value) in row.iter().enumerate() {
113             if max[j] - min[j] == 0.0 {
114                 x_norm[i][j] = 0.0; // Avoid division by zero
115             } else {
116                 // Normalize to range [-1, 1]
117                 x_norm[i][j] = -1.0 + 2.0 * (value - min[j]) / (max[j] - min[j]);
118             }
119         }
120     }
121
122     x_norm

```

```

123 }
124
125 /// Display data in a table.
126 fn table_data(x: &Vec<Vec<f32>>, y_t: &Vec<i32>) {
127     let mut table = Vec::new();
128
129     // Add data to table
130     for (i, row) in x.iter().enumerate() {
131         let mut row_data = Vec::new();
132         row_data.push(y_t[i].to_string());
133         for &value in row.iter() {
134             row_data.push(value.to_string());
135         }
136         table.push(row_data);
137     }
138
139     // Add header to table
140     let table = table.table().title(vec![
141         "y_t".cell().bold(true),
142         "alcohol".cell().bold(true),
143         "malic acid".cell().bold(true),
144         "ash".cell().bold(true),
145         "alkalinity\nof ash".cell().bold(true),
146         "magnesium".cell().bold(true),
147         "total\nphenols".cell().bold(true),
148         "flavanoids".cell().bold(true),
149         "nonflavanoid\nphenols".cell().bold(true),
150         "proanthocyanins".cell().bold(true),
151         "color\nintensity".cell().bold(true),
152         "hue".cell().bold(true),
153         "od280 od315\nof diluted\nwines".cell().bold(true),
154         "proline".cell().bold(true),
155     ]);
156
157     print_stdout(table).unwrap();
158 }
159
160 /// Save `x` and `y_t` to disk in pickle format.
161 fn dump_to_pkl(x: Vec<Vec<f32>>, y_t: Vec<i32>, prefix: &str) {
162     // Create BTreeMaps to serialize
163     let mut x_map = BTreeMap::new();
164     let mut y_map = BTreeMap::new();
165     x_map.insert(String::from("x"), x);
166     y_map.insert(String::from("y_t"), y_t);
167
168     // Serialize to pickle format
169     let x_serialized = serde_pickle::to_vec(&x_map, Default::default()).unwrap();
170     let y_serialized = serde_pickle::to_vec(&y_map, Default::default()).unwrap();
171
172     // Save to disk
173     std::fs::write(format!("./data/x-{}.pkl", prefix), &x_serialized).unwrap();
174     std::fs::write(format!("./data/y_t-{}.pkl", prefix), &y_serialized).unwrap();
175 }
176
177 /// Split data into training and testing sets.
178 use rand::seq::SliceRandom;
179 use rand::thread_rng;
180
181 fn split_data(
182     x: Vec<Vec<f32>>,
183     y_t: Vec<i32>,
184 ) → (Vec<Vec<f32>>, Vec<i32>, Vec<Vec<f32>>, Vec<i32>) {

```

```

185 let class_counts = get_class_count(&y_t);
186 let mut x_train = Vec::new();
187 let mut y_t_train = Vec::new();
188 let mut x_test = Vec::new();
189 let mut y_t_test = Vec::new();
190
191 let wines = x.iter().zip(y_t.iter()).collect::<Vec<_>>();
192
193 for (class, count) in class_counts.iter() {
194     let num_train = (count * 85) / 100;
195
196     let mut class_wines = wines
197         .iter()
198         .filter(|(_, &y)| y == *class)
199         .collect::<Vec<_>>();
200     class_wines.shuffle(&mut thread_rng());
201
202     let (test, train) = class_wines.split_at(num_train as usize);
203
204     for (x, y) in train.into_iter() {
205         x_train.push((*x).clone());
206         y_t_train.push((*y).clone());
207     }
208
209     for (x, y) in test.iter() {
210         x_test.push((*x).clone());
211         y_t_test.push((*y).clone());
212     }
213 }
214
215 (x_train, y_t_train, x_test, y_t_test)
216 }
217
218 fn get_class_count(y_t: &Vec<i32>) -> BTreeMap<i32, i32> {
219     let mut class_counts = BTreeMap::new();
220     for &class in y_t.iter() {
221         *class_counts.entry(class).or_insert(0) += 1;
222     }
223
224     class_counts
225 }

```

Program 7: *prepare\_data.rs*

```

1 use burn::data::dataset::{Dataset, InMemDataset};
2 use serde::{Deserialize, Serialize};
3 use std::collections::BTreeMap;
4 use std::fs::File;
5 use std::io;
6
7 #[derive(Debug, PartialEq, Clone, Serialize, Deserialize)]
8 pub struct WineItem {
9     #[serde(rename = "class")]
10     pub class: i32,
11
12     #[serde(rename = "Alcohol")]
13     pub alcohol: f32,
14
15     #[serde(rename = "Malic acid")]
16     pub malic_acid: f32,
17
18     #[serde(rename = "Ash")]
19     pub ash: f32,

```





```

20
21 #[serde(rename = "Alcalinity of ash")]
22 pub alcalinity_of_ash: f32,
23
24 #[serde(rename = "Magnesium")]
25 pub magnesium: f32,
26
27 #[serde(rename = "Total phenols")]
28 pub total_phenols: f32,
29
30 #[serde(rename = "Flavanoids")]
31 pub flavanoids: f32,
32
33 #[serde(rename = "Nonflavanoid phenols")]
34 pub nonflavanoid_phenols: f32,
35
36 #[serde(rename = "Proanthocyanins")]
37 pub proanthocyanins: f32,
38
39 #[serde(rename = "Color intensity")]
40 pub color_intensity: f32,
41
42 #[serde(rename = "Hue")]
43 pub hue: f32,
44
45 #[serde(rename = "OD280/OD315 of diluted wines")]
46 pub od280_od315_of_diluted_wines: f32,
47
48 #[serde(rename = "Proline")]
49 pub proline: f32,
50 }
51
52 pub struct WineDataset {
53     pub(crate) dataset: InMemDataset<WineItem>,
54 }
55
56 impl WineDataset {
57     pub fn subset(&self, indices: &[usize]) → Self {
58         let data: Vec<WineItem> = indices
59             .iter()
60             .map(|&i| self.dataset.get(i).unwrap().clone())
61             .collect();
62         Self {
63             dataset: InMemDataset::new(data),
64         }
65     }
66 }
67
68 impl WineDataset {
69     pub fn from_pkl(split: &str) → Result<Self, io::Error> {
70         let x_file = File::open(format!("./data/x_{}.pkl", split))?;
71         let y_file = File::open(format!("./data/y_t_{}.pkl", split))?;
72
73         let x_map: BTreeMap<String, Vec<Vec<f32>>> =
74             serde_pickle::from_reader(x_file, Default::default())
75                 .map_err(|e| io::Error::new(io::ErrorKind::InvalidData, e))?;
76         let y_map: BTreeMap<String, Vec<i32>> =
77             serde_pickle::from_reader(y_file, Default::default())
78                 .map_err(|e| io::Error::new(io::ErrorKind::InvalidData, e))?;
79
80         let mut data = Vec::new();
81

```

```

82     for (key, x_values) in x_map.get("x").unwrap().iter().enumerate() {
83         if let Some(y_values) = y_map.get("y_t").unwrap().get(key) {
84             let wine = WineItem {
85                 class: *y_values,
86                 alcohol: x_values[0],
87                 malic_acid: x_values[1],
88                 ash: x_values[2],
89                 alcalinity_of_ash: x_values[3],
90                 magnesium: x_values[4],
91                 total_phenols: x_values[5],
92                 flavanoids: x_values[6],
93                 nonflavanoid_phenols: x_values[7],
94                 proanthocyanins: x_values[8],
95                 color_intensity: x_values[9],
96                 hue: x_values[10],
97                 od280_od315_of_diluted_wines: x_values[11],
98                 proline: x_values[12],
99             };
100             data.push(wine);
101         }
102     }
103
104     let dataset = InMemDataset::new(data);
105
106     Ok(Self { dataset })
107 }
108
109 pub fn train() → Self {
110     Self::from_pkl("train").unwrap()
111 }
112
113 pub fn test() → Self {
114     Self::from_pkl("test").unwrap()
115 }
116 }
117
118 impl Dataset<WineItem> for WineDataset {
119     fn get(&self, index: usize) → Option<WineItem> {
120         self.dataset.get(index)
121     }
122
123     fn len(&self) → usize {
124         self.dataset.len()
125     }
126 }

```

Program 8: dataset.rs

```

1  use crate::dataset::WineItem;
2  use burn::data::data_loader::batcher::Batcher;
3  use burn::prelude::{Backend, Int, Tensor};
4
5  #[derive(Clone)]
6  pub struct WineBatcher<B: Backend> {
7      device: B::Device,
8  }
9
10 impl<B: Backend> WineBatcher<B> {
11     pub fn new(device: B::Device) → Self {
12         Self { device }
13     }
14 }
15

```



```

16 #[derive(Clone, Debug)]
17 pub struct WineBatch<B: Backend> {
18     pub x: Tensor<B, 2>,
19     pub y: Tensor<B, 1, Int>,
20 }
21
22 impl<B: Backend> Batcher<WineItem, WineBatch<B>> for WineBatcher<B> {
23     fn batch(&self, data: Vec<WineItem>) → WineBatch<B> {
24         let x = data
25             .iter()
26             .map(|wine| {
27                 Tensor::<B, 2>::from_data(
28                     [[
29                         wine.alcohol,
30                         wine.malic_acid,
31                         wine.ash,
32                         wine.alcalinity_of_ash,
33                         wine.magnesium,
34                         wine.total_phenols,
35                         wine.flavanoids,
36                         wine.nonflavanoid_phenols,
37                         wine.proanthocyanins,
38                         wine.color_intensity,
39                         wine.hue,
40                         wine.od280_od315_of_diluted_wines,
41                         wine.proline,
42                     ]],
43                     &self.device,
44                 )
45             })
46             .collect::<Vec<_>>();
47
48         let y = data
49             .iter()
50             .map(|wine| Tensor::<B, 1, Int>::from_data([wine.class], &self.device))
51             .collect::<Vec<_>>();
52
53         let x = Tensor::<B, 2>::cat(x, 0).to_device(&self.device);
54         let y = Tensor::<B, 1, Int>::cat(y, 0).to_device(&self.device);
55
56         WineBatch { x, y }
57     }
58 }

```

Program 9: `batcher.rs`

```

1 use crate::batcher::WineBatch;
2 use burn::config::Config;
3 use burn::module::Module;
4 use burn::nn::loss::CrossEntropyLossConfig;
5 use burn::nn::{Dropout, DropoutConfig, Linear, LinearConfig, Relu};
6 use burn::prelude::{Backend, Tensor};
7 use burn::tensor::backend::AutodiffBackend;
8 use burn::tensor::Int;
9 use burn::train::{ClassificationOutput, TrainOutput, TrainStep, ValidStep};
10
11 #[derive(Module, Debug)]
12 pub struct WineModel<B: Backend> {
13     input_layer: Linear<B>,
14     hidden_layer: Linear<B>,
15     output_layer: Linear<B>,
16     activation: Relu,
17     dropout: Dropout,

```



```

18 }
19
20 impl<B: Backend> WineModel<B> {
21     pub fn forward(&self, x: Tensor<B, 2>) → Tensor<B, 2> {
22         let x = self.input_layer.forward(x);
23         let x = self.activation.forward(x);
24         let x = self.dropout.forward(x);
25
26         self.output_layer.forward(x)
27     }
28 }
29
30 impl<B: Backend> WineModel<B> {
31     pub fn forward_classification(
32         &self,
33         x: Tensor<B, 2>,
34         y: Tensor<B, 1, Int>,
35     ) → ClassificationOutput<B> {
36         let output = self.forward(x);
37         let loss = CrossEntropyLossConfig::new()
38             .init(&output.device())
39             .forward(output.clone(), y.clone());
40
41         ClassificationOutput::new(loss, output, y)
42     }
43 }
44
45 impl<B: AutodiffBackend> TrainStep<WineBatch<B>, ClassificationOutput<B>> for WineModel<B> {
46     fn step(&self, batch: WineBatch<B>) → TrainOutput<ClassificationOutput<B>> {
47         let item = self.forward_classification(batch.x, batch.y);
48
49         TrainOutput::new(self, item.loss.backward(), item)
50     }
51 }
52
53 impl<B: Backend> ValidStep<WineBatch<B>, ClassificationOutput<B>> for WineModel<B> {
54     fn step(&self, batch: WineBatch<B>) → ClassificationOutput<B> {
55         self.forward_classification(batch.x, batch.y)
56     }
57 }
58
59 #[derive(Config, Debug)]
60 pub struct WineModelConfig {
61     num_classes: usize,
62     hidden_size: usize,
63     #[config(default = "0.5")]
64     dropout: f64,
65 }
66
67 impl WineModelConfig {
68     pub fn init<B: Backend>(&self, device: &B::Device) → WineModel<B> {
69         WineModel {
70             input_layer: LinearConfig::new(13, self.hidden_size).init(device),
71             hidden_layer: LinearConfig::new(self.hidden_size, self.hidden_size).init(device),
72             output_layer: LinearConfig::new(self.hidden_size, self.num_classes).init(device),
73             activation: Relu::new(),
74             dropout: DropoutConfig::new(self.dropout).init(),
75         }
76     }
77 }

```

Program 10: `model.rs`

```

1 use crate::batcher::WineBatcher;
2 use crate::dataset::WineDataset;
3 use crate::model::WineModelConfig;
4 use burn::config::Config;
5 use burn::data::data_loader::DataLoaderBuilder;
6 use burn::module::Module;
7 use burn::optim::AdamConfig;
8 use burn::record::CompactRecorder;
9 use burn::tensor::backend::AutodiffBackend;
10 use burn::train::metric::{AccuracyMetric, LossMetric};
11 use burn::train::LearnerBuilder;
12
13 #[derive(Config)]
14 pub struct TrainingConfig {
15     pub model: WineModelConfig,
16     pub optimizer: AdamConfig,
17     #[config(default = 4000)]
18     pub num_epochs: usize,
19     #[config(default = 256)]
20     pub batch_size: usize,
21     #[config(default = 1)]
22     pub num_workers: usize,
23     #[config(default = 42)]
24     pub seed: u64,
25     #[config(default = 1e-4)]
26     pub learning_rate: f64,
27 }
28
29 pub fn train<B: AutodiffBackend>(artifact_dir: &str, config: TrainingConfig, device: B::Device) {
30     create_artifact_dir(artifact_dir);
31     config
32         .save(format!("{artifact_dir}/config.json"))
33         .expect("Config should be saved successfully");
34
35     B::seed(config.seed);
36
37     let train_batch = WineBatcher::<B>::new(device.clone());
38     let test_batch = WineBatcher::<B::InnerBackend>::new(device.clone());
39
40     let data_loader_train = DataLoaderBuilder::new(train_batch)
41         .batch_size(config.batch_size)
42         .shuffle(config.seed)
43         .num_workers(config.num_workers)
44         .build(WineDataset::train());
45
46     let data_loader_test = DataLoaderBuilder::new(test_batch)
47         .batch_size(config.batch_size)
48         .shuffle(config.seed)
49         .num_workers(config.num_workers)
50         .build(WineDataset::test());
51
52     let learner = LearnerBuilder::new(artifact_dir)
53         .metric_train_numeric(AccuracyMetric::new())
54         .metric_valid_numeric(AccuracyMetric::new())
55         .metric_train_numeric(LossMetric::new())
56         .metric_valid_numeric(LossMetric::new())
57         .with_file_checkpoint(CompactRecorder::new())
58         .devices(vec![device.clone()])
59         .num_epochs(config.num_epochs)
60         .summary()
61         .build(
62             config.model.init::<B>(&device),

```

```

63     config.optimizer.init(),
64     config.learning_rate,
65 );
66
67 let model_trained = learner.fit(dataloader_train, dataloader_test);
68
69 model_trained
70   .save_file(format!("{artifact_dir}/model"), &CompactRecorder::new())
71   .expect("Trained model should be saved successfully");
72 }
73
74 fn create_artifact_dir(artifact_dir: &str) {
75     // Remove existing artifacts before to get an accurate learner summary
76     std::fs::remove_dir_all(artifact_dir).ok();
77     std::fs::create_dir_all(artifact_dir).ok();
78 }

```

Program 11: `training.rs`

```

1  use burn::backend::cuda_jit::CudaDevice;
2  use burn::backend::{CudaJit};
3  use burn::data::dataset::Dataset;
4  use burn::{
5      backend::{Autodiff, Wgpu},
6      optim::AdamConfig,
7  };
8  use si_project::dataset::WineDataset;
9  use si_project::model::WineModelConfig;
10 use si_project::training::{train, TrainingConfig};
11
12 fn main() {
13     // type MyBackend = Wgpu<f32, i32>;
14     type MyBackend = CudaJit<f32, i32>;
15     type MyAutodiffBackend = Autodiff<MyBackend>;
16
17     // let device = burn::backend::wgpu::WgpuDevice::default();
18     let device = CudaDevice::default();
19
20     let artifact_dir = "./artifacts";
21     train::<MyAutodiffBackend>(
22         artifact_dir,
23         TrainingConfig::new(WineModelConfig::new(3, 64), AdamConfig::new()),
24         device.clone(),
25     );
26 }

```

Program 12: `main.rs`