Zero waste AI: Kompresja i optymalizacja sieci neuronowej

Aleksandra Śliwska, Adam Niewczas, Bartosz Gawron

June 24, 2025

Kod: https://github.com/aSliwska/ssn-projekt/tree/main

1 Metody kompresji i optymalizacji

Uczenie sieci neuronowych wiąże się z eksperymentacją oraz redundancją w nowo wyszkolonych modelach - często są one większe lub miejscami bardziej precyzyjne, niż muszą być. Celem projektu była eksploracja sposobów na kompresję i optymalizację sztucznych sieci neuronowych, aby zmniejszyć ich pojemność na dysku, poziom skomplikowania i czas inferencji. Zbadano trzy popularne techniki - destylację, pruning i kwantyzację.

1.1 Destylacja

Destylacja polega na uczeniu mniejszego modelu (studenta) przy pomocy większego modelu (nauczyciela). Jej celem jest zmniejszenie rozmiaru modelu poprzez uproszczenie jego architektury bez większej straty dokładności. Jest to możliwe, ponieważ duże modele często nie wykorzystują w pełni swojej pojemności.

W tradycyjnym procesie destylacji rozróżniamy wyjścia modelu na „twarde" i „miękkie". Twarde - czyli jedna, najbardziej prawdopodobna klasa - otrzymywana jest poprzez użycie na miękkim wyjściu modelu funkcji softmax. Miękkie wyjście to lista prawdopodobieństw dla każdej klasy.

Mniejsze modele posiadają gorszą zdolność uczenia się - mogą zaabsorbować mniej informacji, więc istnieje możliwość, że same nie będą się w stanie nauczyć zwięzłej reprezentacji wiedzy, zanim zaczną się przeuczać. Destylacja polega na wspomaganiu procesu uczenia studenta poprzez przekazywanie mu pseudoprawdopodobieństw zakodowanych w miękkim wyjściu nauczyciela. Innymi słowy - jest to uczenie poprzez wskazywanie studentowi, jakie wyniki powinien był zwrócić dla wszystkich klas - a nie tylko, która klasa była poprawna. Student jest uczony relacji pomiędzy klasami - ich wspólnych cech.

Funkcja straty w destylacji liczona jest na podstawie dwóch innych:

- twarda strata - standardowa, cross-entropy, liczona z twardego wyjścia studenta i prawdziwej etykiety;
- miękka strata - liczona z miękkiego wyjścia studenta oraz nauczyciela - różnica między dystrybucjami prawdopodobieństwa.

Jeżeli nauczyciel jest bardzo pewny w swoich przewidywaniach - jego miękkie wyjście charakteryzuje się małą entropią (jeden wynik bliski 1, reszta bliska 0) - nie zwraca on wartościowych informacji do destylacji (podobnymi cechami klas). Dlatego w tym procesie używa się także temperatury, która zwiększa entropię prawdopodobieństw.

Opisany powyżej typ destylacji to tylko jeden z wielu. Wyróżnia się:

- Response-based - bierze pod uwagę tylko wyjście modeli (opisane i zaimplementowane w tym projekcie;
- Feature-based - skupia się na aktywacjach w warstwach ukrytych;
- Relation-based - ma na celu przekazanie studentowi całego procesu myślenia, głębszej wiedzy o relacjach między klasami i może być zaimplementowana na różne sposoby.

Innym podziałem destylacji jest rozróżnienie na:

- destylację offline - niezmieniającą modelu nauczyciela podczas uczenia studenta (wykorzystana w tym projekcie);
- destylację online - uczącą równocześnie nauczyciela i studenta;
- samodestylację - w której głębsze warstwy w modelu uczą wierzchnie.

1.2 Pruning

Pruning sieci neuronowych to technika redukcji liczby parametrów (wag) modelu poprzez wyzerowanie mniej istotnych wag. Ma na celu zmniejszenie rozmiaru modelu, przyspieszenie inferencji i ograniczenie zużycia pamięci, bez znacznej utraty jakości. Wyróżniamy dwa główne typy pruningu:

- Pruning niestrukturalny - Wyzerowywanie pojedynczych wag bez narzucenia struktury. Powoduje bardzo rozproszony (sparse) model, który może wymagać specjalnych bibliotek i sprzętu do efektywnej implementacji.
- Pruning strukturalny - Usuwanie całych filtrów, kanałów lub neuronów, dzięki czemu model pozostaje gęsty i może być efektywnie implementowany na standardowych bibliotekach deep learningowych.

1.3 Kwantyzacja

Kwantyzacja to proces redukcji precyzji reprezentacji liczb stosowanych w sztucznych sieciach neuronowych, mający na celu zmniejszenie zapotrzebowania na pamięć obliczeniową oraz przyspieszenie obliczeń. Głównym zastosowaniem tego procesu jest możliwość umieszczenia wytrenowanego modelu na urządzeniach z ograniczonymi zasobami, jak np. smartfony czy mikrokontrolery.

W standardowej formie modele operują na liczbach zmiennoprzecinkowych. Kwantyzacja polega na zamianie tych liczb na całkowite o niższej precyzji, zazwyczaj 8-bitowe, przy jednoczesnym zachowaniu możliwie jak najwyższej dokładności.

Wyróżniamy dwa główne sposoby przeprowadzania kwantyzacji:

- Post-training quantization - kwantyzacja zostaje przeprowadzona po zakończeniu treningu modelu. Dane wejściowe są wykorzystywane do oszacowania zakresów wartości wagi i aktywacji, które następnie są przeskalowywane i zaokrąglane do postaci całkowitoliczbowej.
- Quantization-aware training - kwantyzacja uwzględniana jest już w trakcie uczenia, dzięki czemu model może dostosować się do ograniczeń wynikających z niższej precyzji.

2 Dataset

W projekcie wykorzystano popularny dataset CIFAR-10. To powszechnie stosowany zestaw danych w uczeniu maszynowym, który zawiera łącznie 60 000 kolorowych obrazów o rozdzielczości 32×32 piksele i trzech kanałach (RGB). Zbiór ten podzielony jest na dwie główne części: treningowy (50000 obrazów) i testowy (10000 obrazów), a każde zdjęcie przypisane jest do jednej z 10 klas, takich jak: samolot, samochód, ptak, kot, jeleń, pies, żaba, koń, statek i ciężarówka.

Pomimo niskiej rozdzielczości obrazów, daje on szerokie możliwości testowania architektur, sposobów treningu i technik poprawy jakości (np. augmentacji danych, pruningu, kwantyzacji). Stanowi klasyczny benchmark w literaturze naukowej, pozwala na łatwe porównywanie wyników między różnymi modelami i implementacjami, a dzięki różnorodności klas daje dobry pogląd na zdolność modelu do generalizacji. Ładowanie datasetu odbywa się za pomocą poniższego kodu:

Listing 1: Ładowanie datasetu i ustawienie loader'ów

```
transform = v2.Compose([
    v2.ToImage(),
    v2.ToDtype(torch.float32, scale=True),
    v2.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

```

```
train_dataset = torchvision.datasets.CIFAR10("./../data", train=True,
    transform=transform, download=True)
test_dataset = torchvision.datasets.CIFAR10("./../data", train=False,
    transform=transform, download=True)

train_dataset, validation_dataset =  random_split(train_dataset, [0.8,
    0.2])

print('train set size:', len(train_dataset))
print('validation set size:', len(validation_dataset))
print('test set size:', len(test_dataset))

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True,
    num_workers=num_workers)
validation_loader = DataLoader(validation_dataset, batch_size=64, shuffle=
    True, num_workers=num_workers)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False,
    num_workers=num_workers)

class_names = ["airplane", "automobile", "bird", "cat", "deer", "dog", "
    frog", "horse", "ship", "truck"]
```

Najpierw tworzona jest transformacja, która konwertuje obrazy do tensora, zmienia typ danych na float32 z normalizacją do przedziału [0,1] oraz normalizuje je według typowych dla sieci CNN wartości. Następnie dataset jest pobierany i rozdzielany na train i test. Dodatkowo train_dataset jest rozdzielany na dodatkowy validation_dataset w proporcji 80:20. Każdy zestaw danych jest ładowany do osobnych loaderów poprzez DataLoader, które zapewniają efektywne przetwarzanie danych w batchach, opcjonalne tasowanie i równoległe wczytywanie poprzez ustawienie parametru num_workers. Ostatecznie dostępne są trzy iteratory danych (train_loader, validation_loader i test_loader) gotowe do użycia podczas treningu, walidacji i testowania modelu

3 Model bazowy

Dla odpowiednich zadań zdefiniowano osobne modele bazowe, aby późniejsze operacje na nich wyraźnie pokazywały efekty. Każdy model jednak był podobną w architekturze siecią konwolucyjną napisaną za pomocą biblioteki Pytorch.

3.1 Destylacja

W celu pokazania efektów destylacji istotnym było zdefiniowanie modelu nauczyciela o dość rozwiniętej architekturze. Musiał on osiągać jak najlepsze wyniki, aby dobrze reprezentować podobieństwa między klasami. Składał się on z typowych bloków dla klasycznych sieci konwolucyjnych - warstw konwolucyjnych, funkcji aktywacji ReLU, warstw poolingowych i dropoutu oraz gęstych warstw klasyfikatora.

Listing 2: Architektura modelu nauczyciela do destylacji

```
class TeacherNN(nn.Module):
    def __init__(self, num_classes=10):
        super(TeacherNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(0.3),

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(0.3),
```

```
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(0.3),
        )
        self.classifier = nn.Sequential(
            nn.Linear(128*4*4, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Dla porównania wyników destylacji z normalnym procesem uczenia, nauczono także model studenta bez udziału nauczyciela. Jego architektura musiała być wyraźnie uboższa.

Listing 3: Architektura modelu studenta do destylacji

```
class StudentNN(nn.Module):
    def __init__(self, num_classes=10):
        super(StudentNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(0.2),

            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(0.2),

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(0.2),
        )
        self.classifier = nn.Sequential(
            nn.Linear(64*4*4, 128),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(128, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

3.2 Kwantyzacja

Zdefiniowano architekturę sieci neuronowej w klasie BaseNN, bazującej na warstwach konwolucyjnych (Conv2d) z normalizacją (BatchNorm) i funkcją aktywacji RelU oraz klasyfikatorze w postaci w pełni połączonych warstw (Linear). W celu umożliwienia kwantyzacji dynamicznej oraz statycznej, w modelu wprowadzono dwa specjalne moduły:

- self.quant = torch.quantization.QuantStub() - moduł odpowiedzialny za początkowe przekształcenie danych wejściowych do formatu całkowitoliczbowego.
- self.dequant = torch.quantization.DeQuantStub() - moduł, który przywraca dane do formatu zmiennoprzecinkowego przed końcową predykcją.

Listing 4: Architektura modelu bazowego do kwantyzacji

```
class BaseNN(nn.Module):
    def __init__(self, num_classes=10):
        super(BaseNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Linear(2048, 512),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(512, num_classes)
        )

        self.quant = torch.quantization.QuantStub()
        self.dequant = torch.quantization.DeQuantStub()
```

W metodzie forward dane wejściowe są przepuszczane najpierw przez warstwę QuantStub, następnie przez pozostałe warstwy modelu, a na końcu przez DeQuantStub, dzięki czemu możliwe było przeprowadzenie pełnej kwantyzacji modelu.

Listing 5: Metoda forward modelu bazowego do kwantyzacji

```
def forward(self, x):
        x = self.quant(x)
        x = self.features(x)
        x = x.reshape(x.size(0), -1)
        x = self.classifier(x)
        x = self.dequant(x)
        return x
```

4 Implementacja

4.1 Destylacja

Do nauki nauczyciela i studenta dobrano odpowiednie hiperparametry:

- funkcja straty - CrossEntropyLoss,

- optymalizator - Adam z learning rate 0.001,
- dokładność jako metrykę sprawdzaną w walidacji.

Przed definiowaniem każdego modelu ustawiano to samo ziarno losowości, aby wylosowały się te same wagi domyślne zarówno dla studenta uczącego się samemu, jak i z nauczycielem.

Pętla ucząca dla procesu destylacji różniła się od standardowego uczenia specjalnym sposobem na obliczanie funkcji straty. Najpierw należało wykonać przewidywanie przez nauczyciela i studenta, a następnie dla obu obliczyć softmax (nie był on liczony przez sam model).

Obliczano dwie osobne straty:

- teaching_loss - miękką stratę,
- self_teaching_loss - twardą stratę.

Waga miękkiej straty wynosiła 0.8, a temperatura 2.

Listing 6: Pętla ucząca dla procesu destylacji.

```
for inputs, labels in train_loader:
    inputs, labels = inputs.to(device), labels.to(device)

    optimizer.zero_grad()

    with torch.no_grad():
        teacher_output = teacher(inputs)

    student_output = student(inputs)

    teacher_output_softmax = nn.functional.softmax(teacher_output / T, dim
        =-1)
    student_output_softmax = nn.functional.log_softmax(student_output / T,
        dim=-1)

    teaching_loss = torch.sum(teacher_output_softmax * (
        teacher_output_softmax.log() - student_output_softmax)) /
        student_output_softmax.size()[0] * (T**2)
    self_teaching_loss = ce_loss(student_output, labels)

    loss = teaching_weight * teaching_loss + (1. - teaching_weight) *
        self_teaching_loss

    loss.backward()
    optimizer.step()
```

4.2 Pruning

4.2.1 Unstructured pruning

Na samym początku, przekopiowano bazowy model do nowego obiektu. Pruning niestrukturalny został zaimplementowany za pomocą metody **l1_unstructured**. Działa ona w następujący sposób - Oblicz dla każdego elemntu tensora wag metrykę l1, następnie je posortuj i przypisz maskę tym, które są o zadany procent najmniejsze (w tym przypadku jest to 90%), czyli "przytnij" 90% wag, które są najmniejsze.

Listing 7: Zastosowanie pruningu niestrukturalnego

```
unstructured_model = copy.deepcopy(model_base)
prune.l1_unstructured(unstructured_model.classifier[0], name='weight',
    amount=0.9)
prune.l1_unstructured(unstructured_model.classifier[3], name='weight',
    amount=0.9)

prune.l1_unstructured(unstructured_model.features[0], name='weight', amount
    =0.9)
```

```
prune.l1_unstructured(unstructured_model.features[2], name='weight', amount
    =0.9)
```

Kolejnym krokiem jest fine-tuning sieci. Należy pamiętać aby przyjąć mniejszą liczbę epok i stosownie mniejszy learning-rate, aby wagi mogły się powoli dostosować.

Listing 8: Fine-tuning sieci

```
    trainingEpoch_loss, validationEpoch_loss = train(unstructured_model,
        epochs=7, learning_rate=0.0001)
```

Dodatkowo, do porównania modelu z bazowym, stworzono pomocniczą funkcję do sprawdzania rzadkości (sparsity)

Listing 9: Funkcja do sprawdzania rzadkości modelu

```
    def count_nonzero_parameters(model):
    total_nonzero = 0
    for name, param in model.named_parameters():
        if 'weight' in name:
            total_nonzero += (param != 0).sum().item()
    return total_nonzero
```

Ważnym jest aby zrozumieć, iż samo zastosowanie **l1_unstructured** nie zeruje wag - nakładane są tylko maski, więc należy je usunąć i w ich miejsce, wstawić zerowe wagi.

Listing 10: Usuwanie masek i zasepowanie je faktycznymi zerami

```

# === Usuwanie masek pruningowych ===
for module in [unstructured_model.classifier[0], unstructured_model.
    classifier[3], unstructured_model.features[0], unstructured_model.
    features[2]]:
    prune.remove(module, 'weight')
```

Proces ten umożliwia zapisanie tego modelu za pomocą narzędzi takich jak **onnx** i wykorzystanie go przez inne narzędzia.

Listing 11: Zapisanie modeli jako onnx

```
dummy_input = torch.randn(1, 3, 32, 32, dtype=torch.float32).to(device)

torch.onnx.export(model_base, dummy_input, "model_base.onnx",
                  input_names=["input"], output_names=["output"],
                      opset_version=11,
                  dynamic_axes={"input": {0: "batch_size"}, "output": {0: "
                      batch_size"}})

torch.onnx.export(unstructured_model, dummy_input, "model_sparse.onnx",
                  input_names=["input"], output_names=["output"],
                      opset_version=11,
                  dynamic_axes={"input": {0: "batch_size"}, "output": {0: "
                      batch_size"}})
```

Wykorzystując format onnx, obliczono metrykę i porównano z modelem bazowym. W tym celu stworzono następującą funkcję i zastosowano ją w następujący sposób

Listing 12: Funkcja do metryki

```
    def run_inference_onnx(sess, loader, max_samples=1000):
    times = []
    import time
    for i, (images, labels) in enumerate(loader):
        if i >= max_samples:
            break
        input_array = images.numpy()
        input_name = sess.get_inputs()[0].name
        start = time.time()
        outputs = sess.run(None, {input_name: input_array})
        end = time.time()
```

```
        times.append(end - start)
    avg_time = np.mean(times)
    return avg_time
```

Listing 13: Obliczanie metryk za pomocą onnx

```
sess_base = ort.InferenceSession("model_base.onnx")
sess_sparse = ort.InferenceSession("model_sparse.onnx")

avg_time_base = run_inference_onnx(sess_base, test_loader, max_samples
    =1000)
avg_time_sparse = run_inference_onnx(sess_sparse, test_loader, max_samples
    =1000)
```

4.2.2 Structured pruning

Na samym początku, należy stworzyć funkcję, która będzie obliczać, które warstwy należy usunąć/zostawić w naszym modelu. Funkcja get_pruned_channels wybiera kanały warstwy konwolucyjnej, które warto zachować po pruningu. Najpierw liczy normy L2 wag poszczególnych kanałów, aby ocenić ich „istotność". Następnie, na podstawie podanego procentu (amount), wyznacza liczbę kanałów do usunięcia i zwraca indeksy tych, które mają najwyższe normy, czyli te, które warto pozostawić.

Listing 14: Funkcja ucinająca warstwy w pruningu strukturalnym

```

def get_pruned_channels(conv_layer, amount):
    weight = conv_layer.weight.detach().cpu()
    norms = weight.view(weight.size(0), -1).norm(p=2, dim=1)
    num_prune = int(amount * weight.size(0))
    keep_indices = torch.argsort(norms)[num_prune:]
    return keep_indices.tolist()
```

Faktyczny pruning odbywa się w tym miejscu - obliczane są warstwy, które należy zachować, następnie tworzony jest model z obliczonymi warstwami. W tym przypadku, ucięte zostało 60% kanałów w pierwszej, drugiej i piątej warstwie konwolucyjnej. Dodatkowo, należało sparametryzować model bazowy, dzięki czemu, możliwe będzie tworzenie modelu z zadanymi warstwami

```
keep1 = get_pruned_channels(model_base.features[0], amount=0.9)
keep2 = get_pruned_channels(model_base.features[2], amount=0.9)
keep3 = get_pruned_channels(model_base.features[5], amount=0.9)
structured_model = BaseNN(conv1_out=len(keep1), conv2_out= len(keep2),
    conv3_out=len(keep3)).to(device)
```

Dodatkowo, po zastosowaniu pruningu, trzeba skopiować pozostałe wagi i biasy.

Listing 15: Kopiowanie wag i bias'ów

```

    with torch.no_grad():
    # Conv1
    structured_model.features[0].weight.data = model_base.features[0].
        weight.data[keep1].clone()
    structured_model.features[0].bias.data = model_base.features[0].bias.
        data[keep1].clone()

    # Conv2 (wej cie: keep1, wyj cie: keep2)
    structured_model.features[2].weight.data = model_base.features[2].
        weight.data[keep2][:, keep1, :, :].clone()
    structured_model.features[2].bias.data = model_base.features[2].bias.
        data[keep2].clone()

    # Conv3 (wej cie: keep2, wyj cie: keep3)
    structured_model.features[5].weight.data = model_base.features[5].
        weight.data[keep3][:, keep2, :, :].clone()
    structured_model.features[5].bias.data = model_base.features[5].bias.
        data[keep3].clone()
```

```
14
15      # Klasyfikator
16      structured_model.classifier[0].weight.data = model_base.classifier[0].
            weight.data[:, :len(keep3) * 8 * 8].clone()
17      structured_model.classifier[0].bias.data = model_base.classifier[0].
            bias.data.clone()
```

4.3 Kwantyzacja

4.3.1 Kwantyzacja dynamiczna

W celu zmniejszenia złożoności obliczeniowej modelu oraz przygotowania go do późniejszej kwantyzacji dynamicznej, zastosowano podejście Quantization-Aware Training (QAT).

W implementacji funkcji train proces ten zainicjowany został po zakończeniu dwóch epok treningu modelu:

Listing 16: Inicjacja QAT

```
1 if epoch == 2 and not qat_started:
2             print("Rozpoczynanie Quantization-Aware Training (QAT)...")
3             model.qconfig = torch.quantization.get_default_qat_qconfig("
                  fbgemm")
4             torch.quantization.prepare_qat(model, inplace=True)
5             qat_started = True
```

Na tym etapie:

- model.qconfig przypisuje domyślną konfigurację kwantyzacji dla architektury korzystającej z backendu fbgemm (optymalizowanego pod CPU).
- prepare_qat modyfikuje strukturę modelu, wstawiając symulacje kwantyzacji i dekwantyzacji wewnątrz sieci. Dzięki temu model uczy się, jak zachowywałby się po konwersji wag i aktywacji do mniejszej precyzji.

Dalszy przebieg treningu przebiega standardowo — model jest uczony przy użyciu funkcji straty CrossEntropyLoss oraz optymalizatora Adam. Na koniec każdej epoki obliczane są średnie wartości strat dla zbioru treningowego i walidacyjnego.

Listing 17: Implementacja funkcji train dla kwantyzacji dynamicznej

```
1  def train(model, epochs, learning_rate):
2      trainingEpoch_loss = []
3      validationEpoch_loss = []
4      criterion = nn.CrossEntropyLoss()
5      optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
6
7      qat_started = False
8
9      for epoch in range(epochs):
10         model.train()
11
12         if epoch == 2 and not qat_started:
13             print("Rozpoczynanie Quantization-Aware Training (QAT)...")
14             model.qconfig = torch.quantization.get_default_qat_qconfig("
                   fbgemm")
15             torch.quantization.prepare_qat(model, inplace=True)
16             qat_started = True
17
18         running_loss = 0.0
19
20         for inputs, labels in train_loader:
21             inputs, labels = inputs.to(device), labels.to(device)
22
23             optimizer.zero_grad()
24             outputs = model(inputs)
25             logits = outputs[0] if isinstance(outputs, tuple) else outputs
```

```
            loss = criterion(logits, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        avg_train_loss = running_loss / len(train_loader)
        trainingEpoch_loss.append(avg_train_loss)
        print(f"Epoch {epoch+1}/{epochs}, Training Loss: {avg_train_loss:.4f}")

        # Ewaluacja
        model.eval()
        validation_loss = 0.0
        with torch.no_grad():
            for inputs, labels in validation_loader:
                inputs, labels = inputs.to(device), labels.to(device)

                outputs = model(inputs)
                logits = outputs[0] if isinstance(outputs, tuple) else outputs
                loss = criterion(logits, labels)

                validation_loss += loss.item()

        avg_val_loss = validation_loss / len(validation_loader)
        validationEpoch_loss.append(avg_val_loss)
        print(f"Epoch {epoch+1}/{epochs}, Validation Loss: {avg_val_loss:.4f}")

    return trainingEpoch_loss, validationEpoch_loss
```

Model został wytrenowany przez 7 epok, z czego od trzeciej zastosowano QAT. Po zakończeniu procesu uczenia model został przeniesiony na CPU i poddany właściwej konwersji do wersji kwantyzowanej:

Listing 18: Konwersja do wersji kwantyzowanej

```
model.to("cpu")
model = torch.quantization.convert(model, inplace=True)
```

Funcja torch.quantization.convert przekształca model QAT do wersji rzeczywiście kwantyzowanej. Na tym etapie model jest w pełni zoptymalizowany do wykonywania operacji niskopoziomowych na urządzeniach CPU.

Tak przygotowany model został zapisany do pliku, dzięki czemu możliwe jest późniejsze wczytywanie zakwantyzowanego modelu. Plik .pt zawiera już zredukowane rozmiarowo wagi, zoptymalizowane do szybkiego działania na docelowym sprzęcie.

Listing 19: Zapis modelu do pliku

```
torch.save(model.state_dict(), "../models/quantized_model_during_training.pt")
```

4.3.2 Kwantyzacja statyczna

Na początku wytrenowany został w pełnej precyzji model bazowy gotowy do kwantyzacji. Jego trening trwał 7 epok i został on zapisany do pliku .pt. Następnie, po wczytaniu modelu, przypisano mu konfigurację kwantyzacji przy użyciu backendu fbgemm, analogicznie jak dla kwantyzacji dynamicznej.

Przygotowanie modelu do kwantyzacji przeprowadzono za pomocą funkcji prepare, która wstawia odpowiednie moduły symulujące kwantyzację. Kolejnym krokiem było poddanie modelu kalibracji poprzez przepuszczenie przez niego niewielkiej liczby danych treningowych bez aktualizacji wag. Na podstawie tej operacji warstwy odpowiedzialne za kwantyzację mogły ustawić zakresy wartości

potrzebne do późniejszej reprezentacji w formacie int8.

Ostateczna konwersja do postaci kwantyzowanej nastąpiła poprzez funkcję convert, która zastępuje odpowiednie warstwy ich wersjami niskoprecyzyjnymi. Otrzymany model został zapisany do pliku.

Listing 20: Kwantyzacja statyczna modelu bazowego

```
base_model = BaseNN(num_classes=10).to("cpu")
base_model.load_state_dict(torch.load("../models/quantized_base_model.pt"))

base_model.eval()

base_model.qconfig = torch.quantization.get_default_qconfig('fbgemm')

torch.quantization.prepare(base_model, inplace=True)

with torch.no_grad():
    for i, (inputs, _) in enumerate(train_loader):
        if i >= 10: break
        base_model(inputs)

after_model = torch.quantization.convert(base_model, inplace=False)

torch.save(after_model.state_dict(), "../models/
    quantized_model_after_training.pt")
```

5 Metryki

W celu porównania modelu bazowego i zoptymalizowanego pod względem jakości, rozmiaru oraz czasu działania zastosowano szereg metryk, obejmujących zarówno dokładność klasyfikacji, jak i miary związane z wydajnością obliczeniową:

- Accuracy - podstawowa metryka oceny klasyfikatora, określająca procent poprawnie sklasyfikowanych próbek względem całkowitej liczby przykładów testowych obliczana według wzoru:

$$Accuracy = \frac{Liczba\,poprawnych\,predykcji}{Liczba\,wszytskich\,predykcji} \times 100\%$$

- Macierz pomyłek - narzędzie umożliwiające szczegółową analizę jakości klasyfikacji. Pokazuje, ile przykładów każdej klasy zostało sklasyfikowanych poprawnie, a ile błędnie - zarówno jako inne klasy, jak i całkowicie nietrafione przypadki.
- Model Size - ilość pamięci zajmowanej przez wytrenowany model wyrażana zazwyczaj w megabajtach (MB). Rozmiar zależy głównie od liczby i precyzji wag. Metryka ta ma kluczowe znaczenie w kontekście wdrażania modelu na urządzenia o ograniczonych zasobach.
- FLOPs (Floating Point Operations) - liczba operacji zmiennoprzecinkowych, jakie model musi wykonać podczas jednej inferencji. Służy do oceny złożoności obliczeniowej modelu i ma wpływ na czas działania oraz zużycie energii.
- Latency - czas potrzebny na wykonanie jednej predykcji, mierzony zazwyczaj w milisekundach. Niska latencja jest szczególnie istotna w zastosowaniach czasu rzeczywistego, gdzie decyzje muszą być podejmowane natychmiastowo.
- Energy usage - ilość energii potrzebnej do wykonania jednej predykcji. Metryka szczególnie istotna w kontekście wdrażania modeli na urządzenia przenośne bądź zasilane bateryjnie.

6 Wyniki

6.1 Destylacja

Wyniki uczenia i destylacji prezentują się następująco:

Metryki	Nauczyciel	Samouczący się student	Student
Czas uczenia	2m 17s	37s	3m 53s
Liczba epok	30	19	100
Liczba parametrów	961,706	156,074	156,074
Accuracy	80.62%	73.90%	77.05%
Wielkość modelu	3.67 MB	0.60 MB	0.60 MB
FLOPs	145,792,512	8,802,048	8,802,048
Latency	$0.9395 * 10^{-4}$	$0.8933 * 10^{-4}$	$0.8896 * 10^{-4}$
Energy usage	$8.3338 * 10^{-10}$	$7.9297 * 10^{-10}$	$7.8946 * 10^{-10}$

Table 1: Wyniki uczenia i destylacji.

Liczba epok dla nauczyciela i uczącego się samemu studenta była zdefiniowana przez uczenie aż do momentu, kiedy model zaczął się przeuczać. Student uczący się z nauczycielem nie zaczął się przeuczać po 100 epokach, ale dalsze uczenie dawało mniejszą poprawę (można by go było uczyć dalej). Funkcje straty zobrazowano na poniższych wykresach.

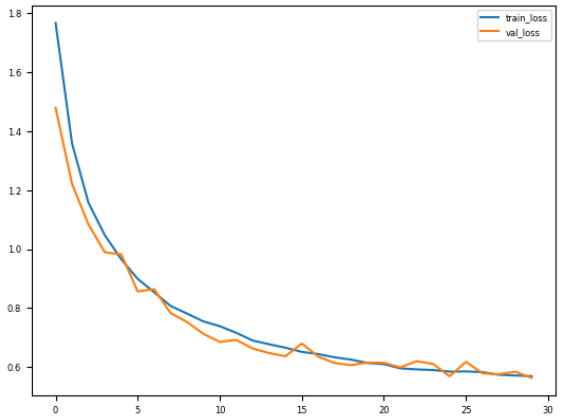

Figure 1: Funkcje straty walidacji i treningu dla nauczyciela.

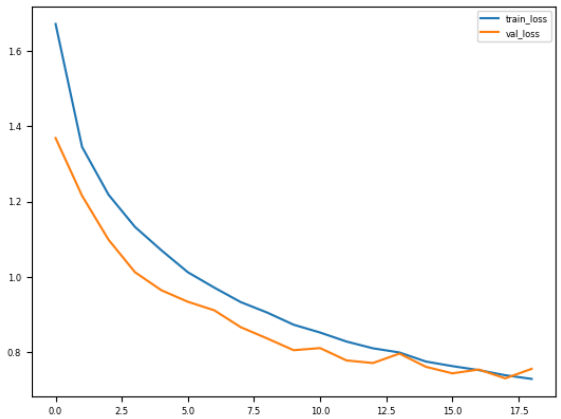

Figure 2: Funkcje straty walidacji i treningu dla samouczącego się studenta.

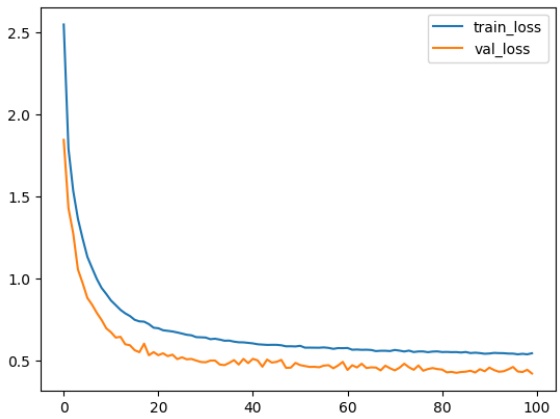

Figure 3: Funkcje straty walidacji i treningu dla studenta.

Ilość parametrów dla obu modeli obliczono w następujący sposób:

Listing 21: Obliczenie liczby parametrów modeli.

```
total_params_student = "{:,}".format(sum(p.numel() for p in model_student.
    parameters()))
```

Emisje obliczano z biblioteką CodeCarbon. W celu uzyskania dokładniejszych wyników obliczono 10 tysięcy predykcji i podzielono wynik przez 10 tysięcy.

Listing 22: Obliczenie emisji.

```
emissionTracker = EmissionsTracker()

emissionTracker.start_task("Test␣teacher")
base_accuracy = test(model_teacher)
teacher_emissions = emissionTracker.stop_task()

emissionTracker.stop()
```

Listing 23: Sposób na obliczenie wielkości modelu na dysku.

```
print(f'Student␣size:␣{os.path.getsize('../models/student2.pt')/(1024*1024)
    :.2f}␣MB')
```

FLOPs obliczono z klasą FlopTensorDispatchMode z biblioteki torchtnt.

Listing 24: Obliczanie FLOPs.

```
def print_pretty_flops(flop_counts, title):
    total_flops = 0
    for module_name in flop_counts.keys():
        operations = flop_counts[module_name]
        module_total = sum(operations.values())
        total_flops += module_total

    print(f"Total␣{title}:␣{total_flops:,.0f}")


def get_flops(model, random_input, name):
    with FlopTensorDispatchMode(model) as ftdm:
        # count forward flops
        res = model(random_input).mean()
        flops_forward = copy.deepcopy(ftdm.flop_counts)

        # reset count before counting backward flops
        ftdm.reset()
        res.backward()
        flops_backward = copy.deepcopy(ftdm.flop_counts)

        print_pretty_flops(flops_forward, "Forward␣FLOPS␣-␣" + name)
```

```
        print_pretty_flops(flops_backward, "Backward FLOPS - " + name)

random_input = torch.randn(1, 3, 32, 32).to(device)

get_flops(model_base, random_input, "Teacher")
```

6.2 Pruning

W porównaniu wyników biorą udział 3 modele:

- model bazowy
- model spruningowany niestrukturalnie
- model spruningowany strukturalnie

i wykorzystano następujące metryki

- Accuracy
- Model size
- FLOPs
- Latency/Inference
- Energy usage

6.2.1 Accuracy

Trafność modelu była oblaczana za pomocą następującej funkcji

Listing 25: Funkcja test do obliczania accuracy

```
    def test(model):
    model.eval()

    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)

            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f"Test Accuracy: {accuracy:.2f}%")
    return accuracy
```

Wyniki prezentują się następująco

Model	Accuracy	Delta
Bazowy	72.11%	-
UP	70.33%	-1.78%
SP	67.67%	-4.44%

Table 2: Porównanie accuracy dla modelu bazowego, QAT i kwantyzacji statycznej.

Trafność modelu różni się zauważalnie w przypadku strukturalnego pruningu. Jednakże, model przy większej ilości epok się przeucza. Niemniej jednak validadion loss jest mniejszy niż w przypadku bazowego modelu, co może oznaczać, że model lepiej generalizuje problem i nie uczy się danych testowych na pamięć. W przypadku Niestrukturalnego pruningu, różnica wynika z przeprowadzonego fine-tuningu, podczas którego, model musiał na nowo dostosować wagi. Jednak różnica nie jest zbyt duża aby się nią przejmować.

6.2.2 Model Size

Do obliczenia rozmiaru pliku wykorzystano następującą metodę:

Listing 26: Obliczanie rozmiaru modeli

```
size_MB_base = os.path.getsize("../models/base.pt") / (1024 * 1024)
size_MB_unstructured = os.path.getsize("../models/unstructured_model.pt") /
    (1024 * 1024)
size_MB_structured = os.path.getsize("../models/structured_model.pt") /
   (1024 * 1024)
```

Wyniki prezentują się następnująco

Model	Model Size (MB)
Bazowy	4.53
UP	4.53
SP	0.91

Table 3: Porównanie rozmiaru zapisanych modeli.

Analizując rozmiar modeli, wynik niestrukturalnego pruningu rzuca się w oczy. Jest on spowodowany tym, że pytorch nie wspiera rzadkich modeli. Dodatkowo, pomimo iż wagi są wyzerowane, to wciąż istnieją w modelu, przez co samo wyzerowanie nie wystarczy by zmiejszyć rozmiar pliku. Może to być przydatne wyedy, gdy dany model będzie korzystać z narzędzi i sprzętu, które potrafią korzystać ze sparsowanych modeli. Jednak ich implementacja jest dość skomplikowana, a niektóre, są niedostępne. Najlepszy wynik osiągnął strukturalny pruning. Ucięcie 90% kanałów poskutkowało drastyczną zmianą w rozmiarze. Osoba pracująca nad modelem musi zdecydować, czy utrata kilku procent tranfności modelu jest do przyjęcia w zamian za zmniejszenie rozmiaru modelu o 80%

6.2.3 FLOPs

Do obliczenia metryki FLOP wykorzystano bibliotekę fvcore w następujący sposób

Listing 27: Obliczanie metryki FLOP

```
# CIFAR-10: wejscie 3x32x32
dummy_input = torch.randn(1, 3, 32, 32).to(device)

flops_per_sample = FlopCountAnalysis(model_base, dummy_input).total()

flops_per_sample = FlopCountAnalysis(unstructured_model, dummy_input).total
    ()

flops_per_sample = FlopCountAnalysis(structured_model, dummy_input).total()
```

Wyniki prezentują się następująco

Model	FLOPs (GB)
Bazowy	94.25
UP	94.25
SP	1.66

Table 4: Porównanie rozmiaru zapisanych modeli.

Sytuacja bardzo podobna do Model Size - structured pruning sprawił, że ilość operacji zmiennoprzecinkowych zmniejszyła sie gwałtownie.

6.2.4 Latency/Inference oraz Energy usage

Metryki te zostały policzone w inny sposób w zależności od modelu - wynikało to z braku możliwości przeprowadzenia sensownej analizy dla nietrukturalnego pruningu, ponieważ pytorch jak i większość narzędzi, nie wspiera rzadkich modeli. Jednak zastosowanie innych narzędzi (onnx), nie przyniosło i tak oczekiwanych skutków.

Dla modelu UP wykorzystano następującą metode obliczenia Inference

```
sess_base = ort.InferenceSession("model_base.onnx")
sess_sparse = ort.InferenceSession("model_sparse.onnx")

avg_time_base = run_inference_onnx(sess_base, test_loader, max_samples
    =1000)
avg_time_sparse = run_inference_onnx(sess_sparse, test_loader, max_samples
    =1000)
```

Wyniki prezentują się następująco

Model	Avg. Inference (s/sample)
Bazowy	0.027
UP	0.029

Table 5: Porównanie rozmiaru zapisanych modeli.

Do obliczenia średniego latency, skorzystano już z tradycyjnej metody mierzenia czasu już dla wszystkich modeli. Dodatkowo, w tym samym czasie, mierzony jest pobór prądu i emisja CO2

Listing 28: Obliczanie latency i Energy usage

```

def measure_model(model, test_loader, device, num_samples=100):
    model.eval()
    model.to(device)

    tracker = EmissionsTracker()
    tracker.start()

    inference_times = []
    with torch.no_grad():
        for i, (images, labels) in enumerate(test_loader):
            if i >= num_samples:
                break
            images = images.to(device)

            start = time.time()
            outputs = model(images)
            end = time.time()

            inference_times.append(end - start)

    emissions = tracker.stop()
    avg_latency = sum(inference_times) / len(inference_times)
    return avg_latency, emissions


# Pomiar dla trzech wersji modelu

device = "cuda" if torch.cuda.is_available() else "cpu"

base_model = BaseNN(num_classes=10).to(device)
base_model.load_state_dict(torch.load("../models/" + "base.pt"))

unstructured_model = BaseNN(num_classes=10).to(device)
unstructured_model.load_state_dict(torch.load("../models/" + "
    unstructured_model.pt"))

structured_model = BaseNN(num_classes=10, conv1_out = len(keep1), conv2_out
     = len(keep2), conv3_out=len(keep3)).to(device)
structured_model.load_state_dict(torch.load("../models/" + "
    structured_model.pt"))

avg_latency_base, emissions_base = measure_model(base_model, test_loader,
    device)
```

```
avg_latency_unstruct, emissions_unstruct = measure_model(unstructured_model
    , test_loader, device)
avg_latency_struct, emissions_struct = measure_model(structured_model,
    test_loader, device)
```

Wyniki prezentują się następująco

Model	Energy usage (kWh)	CO_2 emission (kg)	Avg. Latency s/batch
Bazowy	0.001079	7.14e-04	1.4e-3
UP	0.001085	7.18e-04	1.4e-3
SP	0.001017	6.73e-04	1.3e-3

Table 6: Porównanie zużycia energii zapisanych modeli.

Minimalne różnice są zauważalne znów tylko w przypadku pruningu strukturalnego.

6.3 Kwantyzacja

W celu oceny efektywności przeprowadzonych metod kwantyzacji, porównano trzy wersje modelu:

- model bazowy,
- model kwantyzowany dynamiecznie,
- model kwantyzowany statycznie.

Porównanie przeprowadzono w oparciu o następujące metryki: accuracy, model size, latency oraz energu usage.

6.3.1 Accuracy

W celu obliczenia dokładności klasyfikacji zdefiniowano metodę train, która wylicza accuracy danego modelu na zbiorze testowym.

Listing 29: Metoda train licząca accuracy modelu

```
def test(model):
    correct = 0
    total = 0

    model.eval()

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to("cpu"), labels.to("cpu")

            outputs = model(inputs)

            logits = outputs[0] if isinstance(outputs, tuple) else outputs

            _, predicted = torch.max(logits.data, 1)

            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f"Test Accuracy: {accuracy:.2f}%")
    return accuracy
```

Otrzymane wyniki prezentują się następująco:

Model	Accuracy	Delta
Bazowy	73.11%	-
QAT	74.37%	1.26%
Statyczny	72.68%	-0.43%

Table 7: Porównanie accuracy dla modelu bazowego, QAT i kwantyzacji statycznej.

6.3.2 Model size

Następną obliczoną metryką była wielkość zapisanych modeli, których porównanie przedstawione zostało w poniższej tabeli:

Model	Model Size (MB)
Bazowy	4.54
QAT	1.17
Statyczny	1.17

Table 8: Porównanie rozmiaru zapisanych modeli.

6.3.3 Latency

Do zmierzenia czasu inferencji wykorzystano standardowy moduł time z biblioteki Pythona. Dla każdego z modeli ustawiono model w tryb ewaluacji, wzięto pojedynczy batch z danych testowych, uruchomiono pomiar czasu i wykonano przewidywanie. Zmierzony czas został przeliczony na milisekundy.

Listing 30: Pomiar czasu inferencji

```
import time

base_model.eval()
inputs, labels = next(iter(test_loader))
input = inputs.to("cpu")

start = time.time()
output = base_model(input)
end = time.time()

latency_ms = (end - start) * 1000
```

Otrzymane wyniki prezentują się następująco:

Model	Latency (ms)
Bazowy	344.24
QAT	148.18
Statyczny	142.37

Table 9: Porównanie czasu inferencji zapisanych modeli.

6.3.4 Energy usage

Do oszacowania zużycia energii podczas działania modelu wykorzystano bibliotekę codecarbon, która umożliwia śledzenie emisji CO_2 i poboru energii przez procesy obliczeniowe wykonywane na CPU lub GPU. Zastosowano klasę EmissionsTracker, która działa na zasadzie uruchomienia stopera przed wykonaniem operacji i jego zatrzymania po zakończeniu. Na podstawie czasu działania i używanych zasobów sprzętowych codecarbon dokonuje estymacji zużytej energii elektrycznej oraz odpowiadającej jej emisji CO_2.

Listing 31: Pomiar zużycia energii

```
from codecarbon import EmissionsTracker

tracker = EmissionsTracker()
tracker.start()

output = base_model(input)

emissions = tracker.stop()
```

Porównanie zużycia energii dla zapisanych modeli prezentuje się następująco:

Model	Energy usage (kWh)	CO_2 emission (kg)
Bazowy	0.000039	2.71e-05
QAT	0.000016	1.11e-05
Statyczny	0.000018	1.28e-05

Table 10: Porównanie zużycia energii zapisanych modeli.

7 Wnioski

7.1 Destylacja

Klasyczna destylacja pozwala na znaczne zmniejszenie rozmiaru modelu z odzyskaniem części traconej dokładności podczas upraszczania architektury. Student uczący się sam oraz z nauczycielem jest tego samego rozmiaru - nie ma różnicy w liczbie parametrów, wielkości modelu na dysku ani liczbie FLOPs. Niewiele - lub w ogóle - zmienia się czas inferencji i zużycie energii.

Największe różnice między studentami widać w czasie uczenia i liczbie epok - student uczący się samemu napotyka na „sufit" - szybko zaczyna się przeuczać, nie posiada wystarczającej pojemności, aby samemu nauczyć się generalizować. Traci przez to na dokładności. Z kolei student z nauczycielem jest w stanie uczyć się o wiele dłużej - uczy się wolniej, niż nauczyciel (więcej krótszych epok), ale dochodzi do w miarę podobnej dokładności.

7.2 Pruning

Problem pruningu sieci był realizowany na dwa sposoby - strukturalnie i niestrukturalnie. Do porównania użyto modelu bazowego, purningowego niestrukturalnie i pruningowego struturalnie. Niestety unstructured pruning nie spełniał swojego zadania z racji braku możliwości obsługi rzadkich modeli - pomimo wyzerowanych wag, wciąż model wykonywał obliczenia macierzowe na nich.

Analizując metryki jak i komfort i prostotę użytkowania, najlepiej sprawdził się model z użyciem strukturalnego pruningu - ucięcie kanałów znacząco poprawiło metryki takie jak FLOPs czy jego rozmiar. Utrata kilku procent tranfności modelu może boleć w szczególności w systemach bazujących na medycynie czy zbrojeniach. Należy zdecydować, na czym bardziej zależy programiście.

Opóźnienia, pobór energii czy emisja CO_2 nie mają większej różnicy w stosunku do modeli. Minimalne różnice są jedynie w modelu z zastosowanym pruningiem strukturalnym. Unstructured pruning tylko "zamraża" wagi, przez co fizycznie one dalej istnieją, dlatego ta metoda nie ma wpływu na wyniki.

Podsumowując, realne zastosowanie wraz z efektami posiada metoda zastosowania pruningu strukturalnego. Pruning niestrukturalny przydaje się w sytuacji, gdy mamy dostęp do narzędzi i sprzętu wykorzystujące sparse modele. Należy też dokonać starannego przeszukania dostępnych narzędzi, ponieważ więszkość z nich jest już niedostępna lub nierozwijana. W zależności od tego, jak bardzo programista potrzebuje mniejszego modelu niż jego trafności, powinien dobrać odpowiedni parametr procentowy ucinania kanałów.

7.3 Kwantyzacja

Na podstawie przeprowadzonych eksperymentów porównano ze sobą trzy warianty modelu: model bazowy, model kwantyzowany dynamicznie (QAT) oraz model kwantyzowany statycznie. Analizie poddano kluczowe metryki: accuracy, model size, latency oraz energy usage.

- Wyższa dokładność modelu QAT od modelu bazowego była dość zaskakująca. Może to wynikać z dodatkowego regularizującego efektu kwantyzacji w trakcie treningu, który poprawił uogólnienie modelu.
- Model kwantyzowany statycznie osiągnął nieco gorszą dokładność, co jest spodziewanym skutkiem uproszczenia do formatu całkowitoliczbowego bez wcześniejszej adaptacji modelu.
- W obu przypadkach kwantyzacja pozwoliła znacząco zredukować rozmiar modelu - ponad czterokrotnie w stosunku do modelu bazowego - jednocześnie nie tracąc wiele na dokładności klasyfikacji.

- Zmierzony czas inferencji był znacznie niższy dla modeli zakwantyzowanych - ponad dwukrotnie. Potwierdza to efektywność takich modeli w scenariuszach wymagających niskich opóźnień.
- Podobną zależność zaobserwowano dla zużycia energii - modele zakwantyzowane zużywają dwukrotnie mniej energii od modelu bazowego. Dzięki temu takie modele są atrakcyjnym wyborem dla urządzeń przenośnych czy zasilanych bateryjnie.

Podsumowując, kwantyzacja przynosi wyraźne korzyści w kontekście optymalizacji modeli, umożliwiając redukcję zasobożerności bez istotnych strat jakości predykcji.

8 Źródła

1. https://www.ibm.com/think/topics/knowledge-distillation
2. https://docs.pytorch.org/tutorials/beginner/knowledge_distillation_tutorial.html
3. https://en.wikipedia.org/wiki/Knowledge_distillation
4. https://docs.pytorch.org/tnt/stable/utils/generated/torchtnt.utils.flops.FlopTensorDispatchMo
   html