(A) 🧮 Optional: En enda neuron

- A1: Neuron-implementation: for-loopar och Python-listor (som på tavlan lektion 1), ***alternativt***
- A2: Neuron-implementation: NumPy vektor-multiplikation internt i varje Neuron-objekt

In [1]:
```
def neuron(inputs, weights, bias):
    # Initialize output
    output = 0

    # Calculate output
    for i, w in zip(inputs, weights):
        output += i * w

    # Add bias
    output += bias

    # Apply activation function (ReLU)
    output = max(0, output)

    return output
```

In [2]:
```
import numpy as np

def neuron(inputs, weights, bias):
    # Initialize output
    output = 0

    # Calculate output
    output = np.dot(inputs, weights) + bias

    # Apply activation function (ReLU)
    output = np.maximum(0, output)

    return output
```

(B) ✅ ANN-lager: NumPy version

Det betyder att vi nu inte längre behöver någon klass Neuron, eftersom vi kommer beräkna ett helt lager som en enda stor matris-multiplikation:

- Alla input till ett lager = NumPy-vektor
- Alla vikter för alla neuroner i ett lager = en NumPy-matris
- Observera att vi inte kommer att träna nätverket som är implementerat som en NumPy-beräkning - eftersom det blir mycket enklare i (C) när vi övergår till PyTorch.

In [3]:
```
import numpy as np
```

```python
def layer(inputs, weights, bias):
    # Calculate neuron outputs
    outputs = np.dot(weights, inputs) + bias

    # Apply ReLU to outputs
    outputs = np.maximum(0, outputs)

    return outputs
```

(C) ✅ ANN-lager: PyTorch version:

- Använd PyTorch 2.1 (eller bättre). Använd helst Python 3.10 (eller bättre).
- Kopplas först ihop alla lager i perceptronen så att du får en PyTorch-modell (a.k.a. module). Denna definierar i detalj compute-grafen för din perceptron.
- Använd därefter din perceptron via PyTorch. Googla själv för att få information om hur detta går till rent praktiskt. Det finns gott om information på webben kring PyTorch!
- I denna version ska även träning av nätverket ske, d.v.s. vi ska loopa över epochs, och applicera back-prop. En vidareutveckling av back-prop som kallas ADAM brukar användas eftersom den är både snabb och inte lika ofta fastnar i dåliga lokala minima, jämfört med ren back-prop.
- Se avsnittet "Tips för (C)" nedan.

(D) ✅ Samma som (C), men exekverad på en CUDA GPU

- GPU:n behöver stöda CUDA v11.6 eller högre, vilket motsvarar en GPU från NVIDIA's Pascal-generation eller senare (Exempel på Pascal-kort: GeForce GTX-1080, Quadro P5000, Tesla P100). (Senare generationer: Volta, Turing, Ampère, Ada, Hopper, Blackwell).
- Google Colab har billiga/gratis notebook-instanser med NVIDIA T4 GPU, vilket är en enkel typ av Turing-GPU. Denna fungerar utmärkt för uppgiften, men har du en modern NVIDIA-GPU i din dator är den troligen snabbare än en T4.

In [4]:
```python
import os
import logging
import torch
import torch.nn as nn
import torch.optim as optim
from datetime import datetime
from torch.utils.data import DataLoader, random_split
from torchvision import datasets, transforms

# Define the perceptron neural-network model
class Perceptron(nn.Module):
    # Define the constructor
    def __init__(self):
        super().__init__()
```

```
        # Flatten the input
        self.flatten = nn.Flatten()

        # Define the layers with ReLU activation function
        self.linear_relu_stack = nn.Sequential(
            # Input layer
            nn.Linear(28*28, 512),
            nn.ReLU(),

            # Hidden layer
            nn.Linear(512, 512),
            nn.ReLU(),

            # Output layer
            nn.Linear(512, 10),
        )

    # Define the forward pass
    def forward(self, x):
        # Flatten the input
        x = self.flatten(x)

        # Pass through the layers
        logits = self.linear_relu_stack(x)
        return logits


# Select device to run on
device = torch.accelerator.current_accelerator().type if torch.accelerator.i

# Initialize the model
model = Perceptron().to(device)

# Set hyperparameters
learning_rate = 0.001
num_epochs = 10
batch_size = 64
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Load the MNIST dataset
training_dataset = datasets.MNIST(root='./data', train=True, download=True,
testing_dataset = datasets.MNIST(root='./data', train=False, download=True,

# Split training data into train and validation subsets
training_subset_size = int(0.8 * len(training_dataset))
validation_subset_size = len(training_dataset) - training_subset_size
training_subset, validation_subset = random_split(training_dataset, [trainin

# Create DataLoaders
train_loader = DataLoader(training_subset, batch_size=batch_size, shuffle=Tr
validation_loader = DataLoader(validation_subset, batch_size=batch_size, shu
testing_loader = DataLoader(testing_dataset, batch_size=batch_size, shuffle=

# Create a unique id and directory for the run
```

```
checkpoint_filename_prefix = 'checkpoint'
run_id = datetime.now().strftime("%Y%m%d_%H%M%S")
run_dir = os.path.join('models', f'run_{run_id}')
checkpoints_dir = os.path.join(run_dir, 'checkpoints')
os.makedirs('models', exist_ok=True)
os.makedirs(run_dir, exist_ok=True)
os.makedirs(checkpoints_dir, exist_ok=True)

# Set up logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s
logger = logging.getLogger()
log_file = os.path.join(run_dir, f'run_{run_id}_training.log')
fhandler = logging.FileHandler(filename=log_file, mode='a')
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(me
fhandler.setFormatter(formatter)
logger.addHandler(fhandler)

# Log hyperparameters
logger.info("=" * 100)
logger.info(f"Run ID: {run_id}")
logger.info(f"Training configuration:")
logger.info(f"Learning rate: {learning_rate}")
logger.info(f"Batch size: {batch_size}")
logger.info(f"Epochs: {num_epochs}")
logger.info(f"Optimizer: Adam")
logger.info(f"Loss function: CrossEntropyLoss")

# Training and validation loop
best_val_loss = float('inf')
best_model_path = None
for epoch in range(num_epochs):
    # Training phase
    model.train()
    running_train_loss = 0.0
    for x, y in train_loader:
        # Move data to device
        x, y = x.to(device), y.to(device)

        # Forward pass
        outputs = model(x)
        loss = criterion(outputs, y)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Update running loss
        running_train_loss += loss.item()

    # Calculate average loss
    avg_train_loss = running_train_loss / len(train_loader)

    # Print training loss
    logger.info("="*100)
    logger.info(f"Epoch [{epoch+1}/{num_epochs}]")
```

```
    logger.info(f"Training Loss: {avg_train_loss:.4f}")

    # Validation phase
    model.eval()
    running_val_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for x, y in validation_loader:
            # Move data to device
            x, y = x.to(device), y.to(device)

            # Forward pass
            outputs = model(x)
            loss = criterion(outputs, y)
            running_val_loss += loss.item()

            # Calculate accuracy
            _, predicted = torch.max(outputs, 1)
            total += y.size(0)
            correct += (predicted == y).sum().item()

    # Calculate average loss and accuracy
    avg_val_loss = running_val_loss / len(validation_loader)
    val_accuracy = 100 * correct / total

    # Print validation loss and accuracy
    logger.info(f"Validation Loss: {avg_val_loss:.4f}")
    logger.info(f"Validation Accuracy: {val_accuracy:.2f}%")

    # Save the checkpoint
    checkpoint_filename = f'{checkpoint_filename_prefix}_epoch_{epoch+1}.pth
    checkpoint_path = os.path.join(checkpoints_dir, checkpoint_filename)
    torch.save(model.state_dict(), checkpoint_path)

    # Update the best model if the current model has a lower validation loss
    if avg_val_loss < best_val_loss:
        best_model_path = checkpoint_path # Cache the path to the best model
        best_val_loss = avg_val_loss

# Get the best model for testing
model.load_state_dict(torch.load(best_model_path))

# Testing loop
model.eval()
running_test_loss = 0.0
correct = 0
total = 0
with torch.no_grad():
    for x, y in testing_loader:
        # Move data to device
        x, y = x.to(device), y.to(device)

        # Forward pass
        outputs = model(x)
        loss = criterion(outputs, y)
```

```
        running_test_loss += loss.item()

        # Calculate accuracy
        _, predicted = torch.max(outputs, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()

# Calculate average loss and accuracy
avg_test_loss = running_test_loss / len(testing_loader)
test_accuracy = 100 * correct / total

# Print test loss and accuracy
logger.info("="*100)
logger.info(f"Best Model: {best_model_path}")
logger.info(f"Test Loss: {avg_test_loss:.4f}")
logger.info(f"Test Accuracy: {test_accuracy:.2f}%")
logger.info("="*100)
```

```
2025-04-27 21:24:49,551 - INFO - ===========================================
=========================================================
2025-04-27 21:24:49,552 - INFO - Run ID: 20250427_212449
2025-04-27 21:24:49,552 - INFO - Training configuration:
2025-04-27 21:24:49,552 - INFO - Learning rate: 0.001
2025-04-27 21:24:49,553 - INFO - Batch size: 64
2025-04-27 21:24:49,553 - INFO - Epochs: 10
2025-04-27 21:24:49,554 - INFO - Optimizer: Adam
2025-04-27 21:24:49,554 - INFO - Loss function: CrossEntropyLoss
2025-04-27 21:24:49,552 - INFO - Run ID: 20250427_212449
2025-04-27 21:24:49,552 - INFO - Training configuration:
2025-04-27 21:24:49,552 - INFO - Learning rate: 0.001
2025-04-27 21:24:49,553 - INFO - Batch size: 64
2025-04-27 21:24:49,553 - INFO - Epochs: 10
2025-04-27 21:24:49,554 - INFO - Optimizer: Adam
2025-04-27 21:24:49,554 - INFO - Loss function: CrossEntropyLoss
2025-04-27 21:24:53,737 - INFO - ===========================================
=========================================================
2025-04-27 21:24:53,738 - INFO - Epoch [1/10]
2025-04-27 21:24:53,738 - INFO - Training Loss: 0.2477
2025-04-27 21:24:54,644 - INFO - Validation Loss: 0.1408
2025-04-27 21:24:54,644 - INFO - Validation Accuracy: 95.67%
2025-04-27 21:24:58,749 - INFO - ===========================================
=========================================================
2025-04-27 21:24:58,750 - INFO - Epoch [2/10]
2025-04-27 21:24:58,750 - INFO - Training Loss: 0.0914
2025-04-27 21:24:59,630 - INFO - Validation Loss: 0.1056
2025-04-27 21:24:59,631 - INFO - Validation Accuracy: 96.67%
2025-04-27 21:25:03,755 - INFO - ===========================================
=========================================================
2025-04-27 21:25:03,755 - INFO - Epoch [3/10]
2025-04-27 21:25:03,756 - INFO - Training Loss: 0.0595
2025-04-27 21:25:04,636 - INFO - Validation Loss: 0.0993
2025-04-27 21:25:04,637 - INFO - Validation Accuracy: 97.03%
2025-04-27 21:25:08,646 - INFO - ===========================================
=========================================================
2025-04-27 21:25:08,647 - INFO - Epoch [4/10]
2025-04-27 21:25:08,647 - INFO - Training Loss: 0.0429
2025-04-27 21:25:09,510 - INFO - Validation Loss: 0.1043
2025-04-27 21:25:09,511 - INFO - Validation Accuracy: 96.83%
2025-04-27 21:25:13,515 - INFO - ===========================================
=========================================================
2025-04-27 21:25:13,515 - INFO - Epoch [5/10]
2025-04-27 21:25:13,516 - INFO - Training Loss: 0.0345
2025-04-27 21:25:14,395 - INFO - Validation Loss: 0.0874
2025-04-27 21:25:14,396 - INFO - Validation Accuracy: 97.62%
2025-04-27 21:25:18,404 - INFO - ===========================================
=========================================================
2025-04-27 21:25:18,405 - INFO - Epoch [6/10]
2025-04-27 21:25:18,405 - INFO - Training Loss: 0.0289
2025-04-27 21:25:19,309 - INFO - Validation Loss: 0.0998
2025-04-27 21:25:19,310 - INFO - Validation Accuracy: 97.39%
2025-04-27 21:25:23,418 - INFO - ===========================================
=========================================================
2025-04-27 21:25:23,419 - INFO - Epoch [7/10]
2025-04-27 21:25:23,419 - INFO - Training Loss: 0.0255
```

```
2025-04-27 21:25:24,324 - INFO - Validation Loss: 0.1060
2025-04-27 21:25:24,325 - INFO - Validation Accuracy: 97.44%
2025-04-27 21:25:28,351 - INFO - ===========================================
==========================================================
2025-04-27 21:25:28,351 - INFO - Epoch [8/10]
2025-04-27 21:25:28,352 - INFO - Training Loss: 0.0209
2025-04-27 21:25:29,226 - INFO - Validation Loss: 0.0844
2025-04-27 21:25:29,227 - INFO - Validation Accuracy: 97.97%
2025-04-27 21:25:33,246 - INFO - ===========================================
==========================================================
2025-04-27 21:25:33,247 - INFO - Epoch [9/10]
2025-04-27 21:25:33,247 - INFO - Training Loss: 0.0159
2025-04-27 21:25:34,145 - INFO - Validation Loss: 0.1119
2025-04-27 21:25:34,145 - INFO - Validation Accuracy: 97.62%
2025-04-27 21:25:38,233 - INFO - ===========================================
==========================================================
2025-04-27 21:25:38,233 - INFO - Epoch [10/10]
2025-04-27 21:25:38,234 - INFO - Training Loss: 0.0170
2025-04-27 21:25:39,103 - INFO - Validation Loss: 0.1339
2025-04-27 21:25:39,103 - INFO - Validation Accuracy: 97.19%
2025-04-27 21:25:39,832 - INFO - ===========================================
==========================================================
2025-04-27 21:25:39,833 - INFO - Best Model: models/run_20250427_212449/chec
kpoints/checkpoint_epoch_8.pth
2025-04-27 21:25:39,833 - INFO - Test Loss: 0.0778
2025-04-27 21:25:39,834 - INFO - Test Accuracy: 98.00%
2025-04-27 21:25:39,834 - INFO - ===========================================
==========================================================
```