

Fedorchenko Mikhail Valerevich

Архитектура  
микропроцессора/микроконтроллера  
Microprocessor/microcontroller architecture

Транзисторы  
Transistors

Логические элементы  
(вентили) и Коммутация  
электрических сигналов  
Logic Elements (Gates)  
and Electrical Signal  
Switching

Цифровая схема -  
Логическая схема  
Digital circuit - Logic  
circuit

Микрочип  
Microchip

Полупроводниковый  
материал  
Semiconductor material

Межсоединения  
(проводники)  
Interconnection  
(conductors)

Функциональный  
блок/модуль  
Functional block/module

Микросхема (или  
интегральная схема)  
Microcircuit (or  
integrated circuit)

Программируемая  
логическая интегральная  
схема (PLD) или  
Специализированная  
интегральная схема (ASIC)  
или Система-на-кристалле  
(SoC)  
Programmable Logic Device  
(PLD) or Application Specific  
Integrated Circuit (ASIC) or  
System-on-Chip (SoC)

Fedorchenko Mikhail Valerevich

Подсистема: Реализация и  
Функционирование Слоя Нейронной Сети  
Subsystem: Neural Network Layer  
Implementation and Functioning

Нейроны (или узлы)  
Neurons (or nodes)

Слой нейронной сети  
(Neural Network Layer)

Активный Нейрон  
(Activated Neuron) /  
Нейрон с активацией

Нейросеть  
Neural  
network

Связи (или  
веса)  
Links (or  
weights)

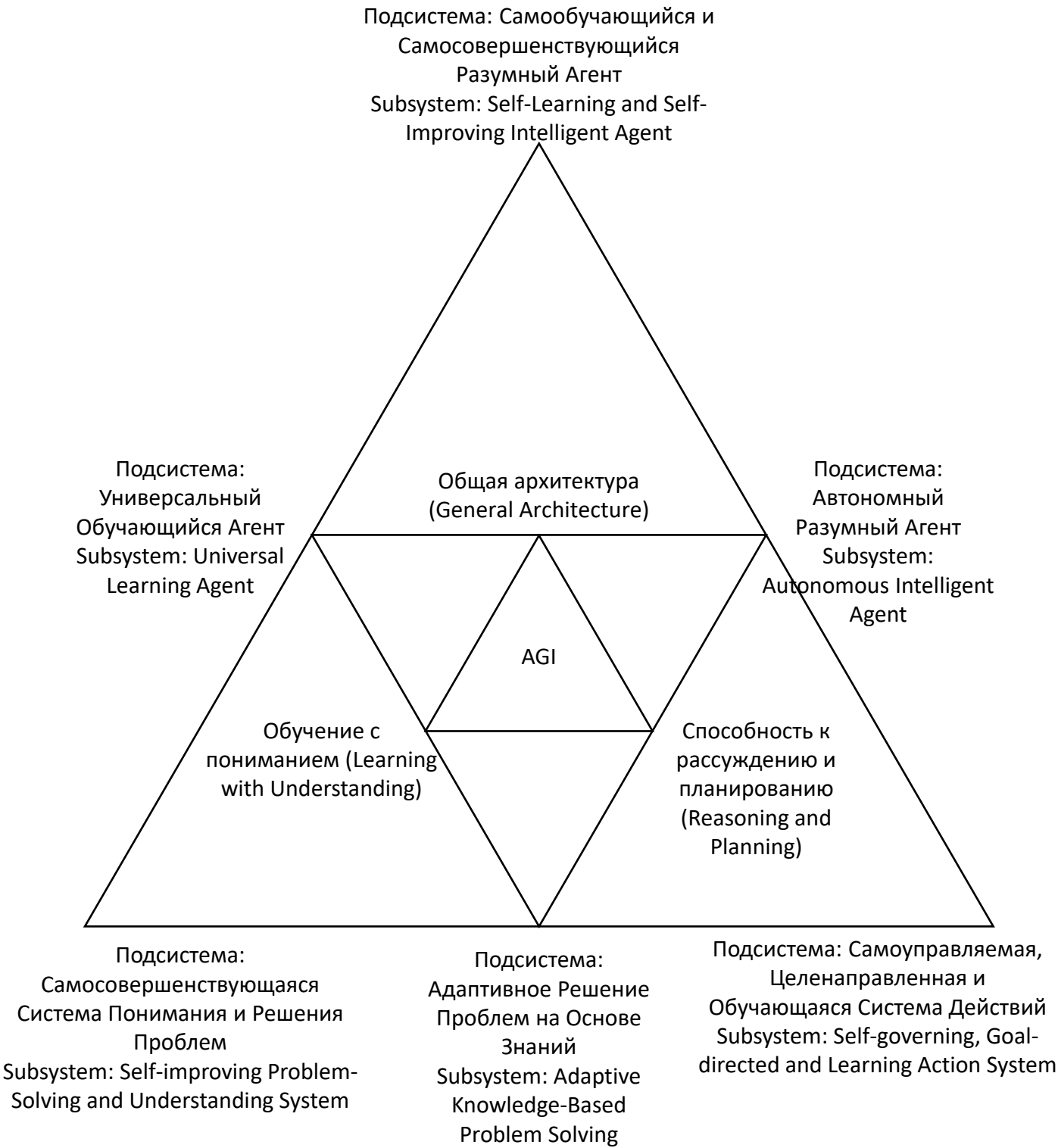
Функция  
активации  
Activation  
function

Подсистема:  
Параллельная  
Обработка Данных в  
Слое Нейронной Сети  
Subsystem: Parallel Data  
Processing in the Neural  
Network Layer

Процесс Обработки  
Данных (Data Processing  
within a Neuron)

Подсистема: Реализация и  
Модульность Вычислений в  
Нейроне  
Subsystem: Implementation  
and Modularity of  
Computations in Neuron

Fedorchenko Mikhail Valerevich



Fedorchenko Mikhail Valerevich

This code is a simplified version and can be extended depending on specific requirements.

```
import numpy as np
```

```
class Neuron:
```

```
    def __init__(self, weights, activation_function):
        self.weights = weights
        self.activation_function = activation_function
```

```
    def activate(self, inputs):
        total = np.dot(self.weights, inputs)
        return self.activation_function(total)
```

```
class NeuralNetworkLayer:
```

```
    def __init__(self, neurons):
        self.neurons = neurons
```

```
    def process(self, inputs):
        outputs = [neuron.activate(inputs) for neuron in self.neurons]
        return outputs
```

```
class SelfLearningAgent:
```

```
    def __init__(self, neural_network):
        self.neural_network = neural_network
```

```
    def learn(self, data):
        # Простейший пример обучения: обновление весов на основе данных
        for layer in self.neural_network:
            for neuron in layer.neurons:
                neuron.weights += np.random.rand(len(neuron.weights)) * 0.1
```

```
    def act(self, inputs):
        outputs = inputs
        for layer in self.neural_network:
            outputs = layer.process(outputs)
        return outputs
```

```
class AutonomousIntelligentAgent:
```

```
    def __init__(self, learning_agent):
        self.learning_agent = learning_agent
```

```
    def perform_task(self, task_data):
        # Пример выполнения задачи с использованием обученного агента
        result = self.learning_agent.act(task_data)
        return result
```

```
# Пример использования
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

```
# Создание нейронов и слоев нейронной сети
```

```
neurons_layer1 = [Neuron(np.random.rand(2), sigmoid) for _ in range(3)]
neurons_layer2 = [Neuron(np.random.rand(3), sigmoid) for _ in range(2)]
layer1 = NeuralNetworkLayer(neurons_layer1)
layer2 = NeuralNetworkLayer(neurons_layer2)
neural_network = [layer1, layer2]
```

```
# Создание самообучающегося агента
```

```
learning_agent = SelfLearningAgent(neural_network)
```

```
# Обучение агента на данных
```

```
data = np.random.rand(10, 2)
learning_agent.learn(data)
```

```
# Создание автономного интеллектуального агента
```

```
autonomous_agent = AutonomousIntelligentAgent(learning_agent)
```

```
# Выполнение задачи
```

```
task_data = np.random.rand(2)
result = autonomous_agent.perform_task(task_data)
print("Результат выполнения задачи:", result)
```

```

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import random
from collections import deque

```

# Neural Network for the Agent (Core of the AGI)

```

class NeuralNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(NeuralNetwork, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, hidden_size)
        self.layer3 = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.relu(self.layer2(x))
        x = self.layer3(x)
        return x

```

# AGI Agent Class

```

class AGIAgent:
    def __init__(self, state_size, action_size, hidden_size=64, gamma=0.99, lr=0.001, memory_size=10000):
        self.state_size = state_size
        self.action_size = action_size
        self.gamma = gamma # Discount factor for future rewards
        self.lr = lr # Initial learning rate
        self.memory = deque(maxlen=memory_size) # Experience replay memory
        self.model = NeuralNetwork(state_size, hidden_size, action_size)
        self.optimizer = optim.Adam(self.model.parameters(), lr=self.lr)
        self.criterion = nn.MSELoss()
        self.epsilon = 1.0 # For epsilon-greedy exploration
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995

```

# Subsystem: Universal Learning Agent (Learning with Understanding)

```

def learn(self, batch_size):
    if len(self.memory) < batch_size:
        return

    # Sample a batch of experiences
    batch = random.sample(self.memory, batch_size)
    states, actions, rewards, next_states, dones = zip(*batch)

    states = torch.FloatTensor(states)
    actions = torch.LongTensor(actions)
    rewards = torch.FloatTensor(rewards)
    next_states = torch.FloatTensor(next_states)
    dones = torch.FloatTensor(dones)

    # Compute Q-values
    q_values = self.model(states).gather(1, actions.unsqueeze(1)).squeeze(1)
    next_q_values = self.model(next_states).max(1)[0]
    target_q = rewards + (1 - dones) * self.gamma * next_q_values

    # Compute loss and update model
    loss = self.criterion(q_values, target_q.detach())
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    # Subsystem: Self-improving Problem-Solving (Adjust learning rate dynamically)
    self.self_improve()

```

# Subsystem: Autonomous Intelligent Agent (Reasoning and Planning)

```

def reason_and_plan(self, state):
    # Convert state to tensor
    state = torch.FloatTensor(state).unsqueeze(0)

    # Epsilon-greedy action selection
    if random.random() < self.epsilon:
        action = random.randrange(self.action_size)
    else:
        with torch.no_grad():
            q_values = self.model(state)
            action = q_values.max(1)[1].item()

```

```

# Decay epsilon for exploration-exploitation trade-off
if self.epsilon > self.epsilon_min:
    self.epsilon *= self.epsilon_decay

return action

# Subsystem: Self-improving Problem-Solving System
def self_improve(self):
    # Dynamically adjust learning rate based on performance (simplified)
    # If the agent is not improving (e.g., loss isn't decreasing), reduce learning rate
    current_lr = self.optimizer.param_groups[0]['lr']
    if random.random() < 0.1: # Simulate performance check
        new_lr = max(current_lr * 0.9, 1e-5) # Decrease learning rate, with a minimum
        for param_group in self.optimizer.param_groups:
            param_group['lr'] = new_lr
        print(f"Adjusted learning rate to {new_lr}")

# Subsystem: Self-governing, Goal-directed Learning Action System
def act(self, state, env):
    action = self.reason_and_plan(state)
    next_state, reward, done, _ = env.step(action)
    self.memory.append((state, action, reward, next_state, done))
    return next_state, reward, done, action

# Simple Grid World Environment for Testing
class GridWorld:
    def __init__(self, size=5):
        self.size = size
        self.state = [0, 0] # Starting position
        self.goal = [size-1, size-1] # Goal position
        self.actions = [(0, 1), (1, 0), (0, -1), (-1, 0)] # Right, Down, Left, Up

    def reset(self):
        self.state = [0, 0]
        return np.array(self.state)

    def step(self, action):
        # Update position based on action
        move = self.actions[action]
        new_state = [self.state[0] + move[0], self.state[1] + move[1]]

        # Check boundaries
        if 0 <= new_state[0] < self.size and 0 <= new_state[1] < self.size:
            self.state = new_state

        # Compute reward and done
        reward = -1 # Step penalty
        done = False
        if self.state == self.goal:
            reward = 100 # Goal reward
            done = True

        return np.array(self.state), reward, done, {}

# Main Training Loop
def train_agent():
    env = GridWorld(size=5)
    agent = AGIAgent(state_size=2, action_size=4, hidden_size=64)
    episodes = 1000
    batch_size = 32

    for episode in range(episodes):
        state = env.reset()
        total_reward = 0
        done = False

        while not done:
            next_state, reward, done, action = agent.act(state, env)
            total_reward += reward
            state = next_state

        # Learn from experience
        agent.learn(batch_size)

    if episode % 100 == 0:
        print(f"Episode {episode}, Total Reward: {total_reward}, Epsilon: {agent.epsilon:.3f}")

if __name__ == "__main__":
    train_agent()

```