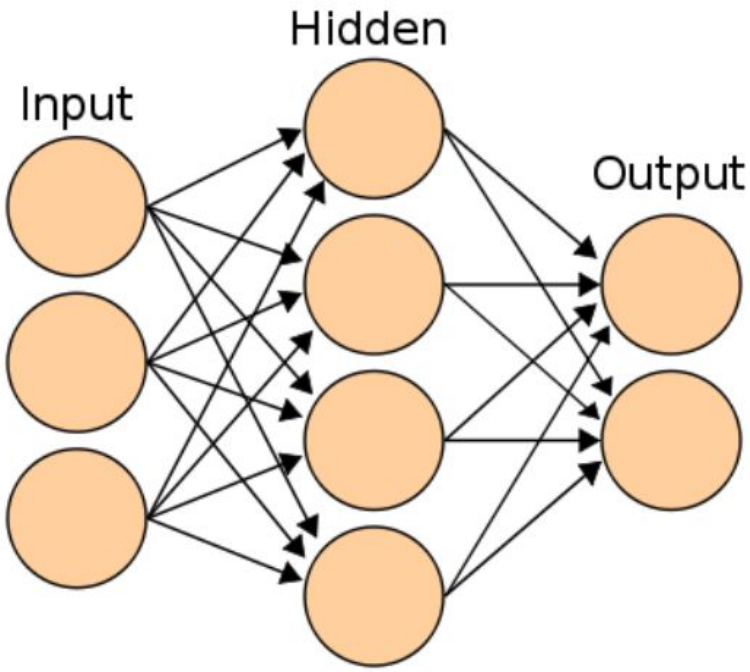
Нейронные сети

Занятие 2
Алгоритм обратного распространения ошибки

Повторение



Если мы уже аппроксимируем какую угодно функцию, зачем что-то еще?

Теорема (универсальный аппроксиматор)[1]

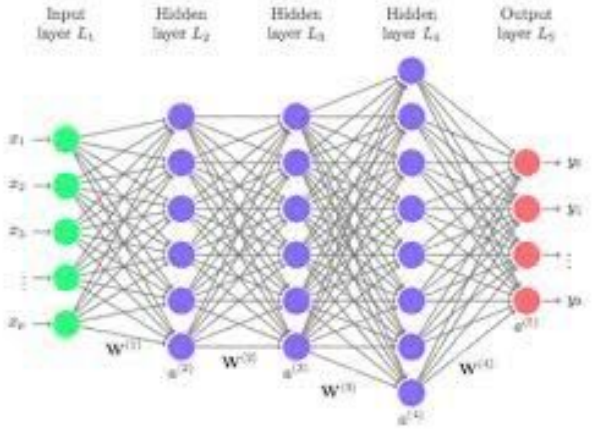
Любую непрерывную на компакте функцию можно равномерно приблизить нейронной сетью с одним скрытым слоем.

Повторение

- Возможно, потребуется очень много нейронов;
- Непонятно, можно ли будет найти отпимум;
- А какая обобщающаю способность? (inductive bias);
  На самом деле, мы хотим делать глубокие сети:

Иерархическая структура извлечения признаков.

Каждый новый слой использует предыдущие признаки, чтобы делать новые.



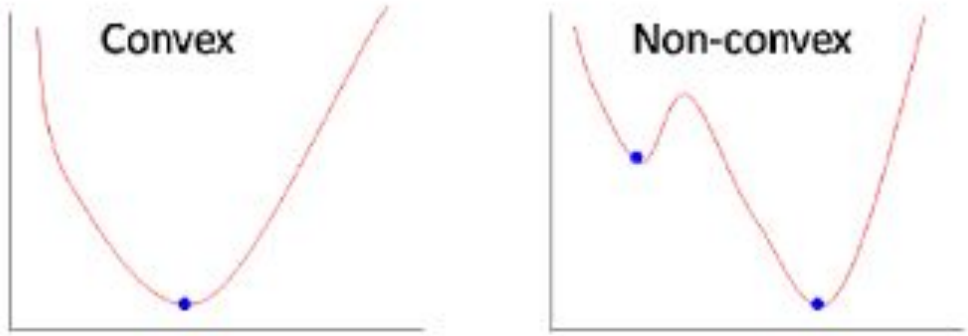
Аугментация

Небольшие преобразования входных данных, которые не меняют таргет.

– Для картинок сдвиги, повороты, кроп и тд
– Для текстов: синонимы, перевод в обе стороны
– Для аудио: добавления шумов, музыки, ускорение аудио и тд

Optimization

- В итоге: $\mathbf{w}^* = argmin_{\mathbf{w}} Q(X, \mathbf{w})$
- Иногда можно руками посчитать
- Можно делать **градиентный** спуск (что это такое?)
- Оптимизация может давать глобальный оптимум;



Градиентный спуск

Антиградиент функции показывает направления наискорейшего убывания функции.

Ищем минимум $Q(w)$

Выбрать начальную длину шага α_0, начальное приближение w_0

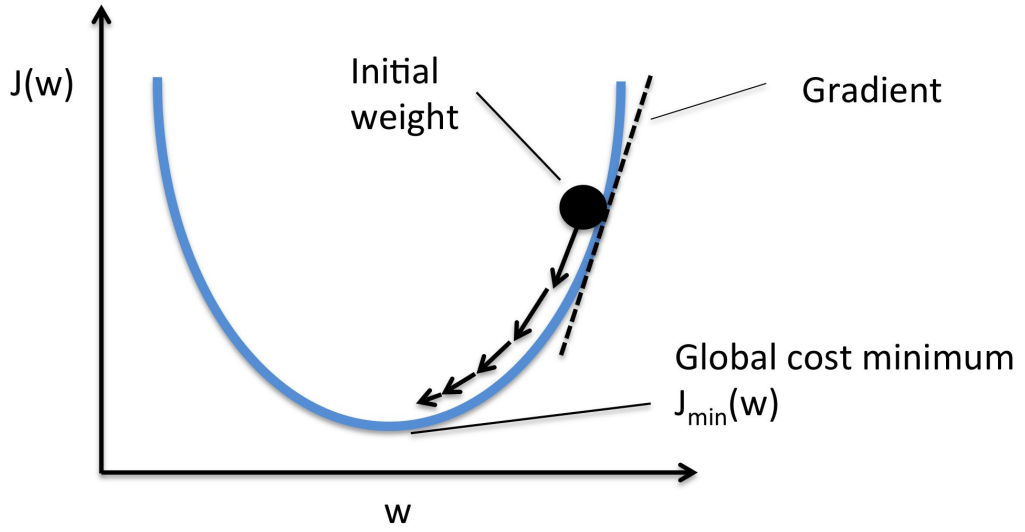
$$w_{new} = w_{old} - \alpha \nabla_w Q(w_{old})$$

$$\alpha = f(k), k = k + 1$$

Повторять (2), (3) до сходимости $Q(w)$ или w

$$f(k) = \alpha_0, f(k) = \frac{\alpha_0}{k}, f(k) = \frac{\alpha_0}{k^p}, \ldots$$

Глобальный минимум или локальный?



Стохастический градиентный спуск

В задачах машинного обучения, оптимизируемая функция имеет специальный вид:

$$Q(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} L(\boldsymbol{w}, \boldsymbol{x_i}, y_i)$$

$$\nabla_w Q(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{w}} L(\boldsymbol{w}, \boldsymbol{x_i}, y_i)$$

Выбрать начальный шаг α_0, начальное приближение $\boldsymbol{w}_0$, размер батча n

Выбрать случайно $\{j_1, i_2, \ldots j_n\}$

Оценить градиент $\nabla_{\boldsymbol{w}} Q^*(w_{old}) = \frac{1}{n}\sum_{j=1}^{n} \nabla_{\boldsymbol{w}} L(\boldsymbol{w_{old}}, \boldsymbol{x_j}, y_j)$

$\boldsymbol{w_{new}} = \boldsymbol{w_{old}} - \alpha \nabla_{\boldsymbol{w}} Q^*(\boldsymbol{w_{old}})$

$\alpha = f(k), k = k + 1$

Повторять (2 - 5) до сходимости

Обычно перемешивают всю выборку случайно.

Когда прошли все выборку, то говорят, что прошла **одна эпоха**

$n = N$ — градиентный спуск (Gradient Descent, GD)

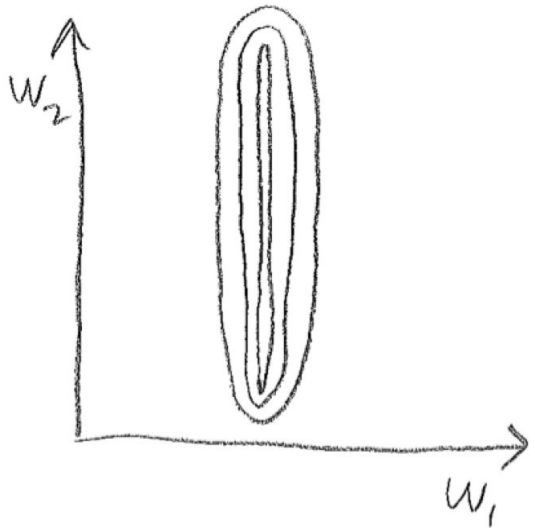
$n = 1$ — стохастический градиентный спуск (Stochastic Gradient Descent, SGD)

$n > 1, n < N$ — мини-батч градиентный спуск (Mini-Batch Gradient Descent, MBGD)

Нормализация данных

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
⋮	⋮	⋮

$$\overline{w_i} = \overline{y}\, x_i$$



$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}$$

The quantity $\lambda_{\max}/\lambda_{\min}$ is known as the condition number of $\mathbf{A}$. Larger condition numbers imply slower convergence of gradient descent.

Источник:
cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec07.pdf

Программа занятия

1. Производные на графе
2. Бэкпроп для полносвязной сети
3. Pytorch

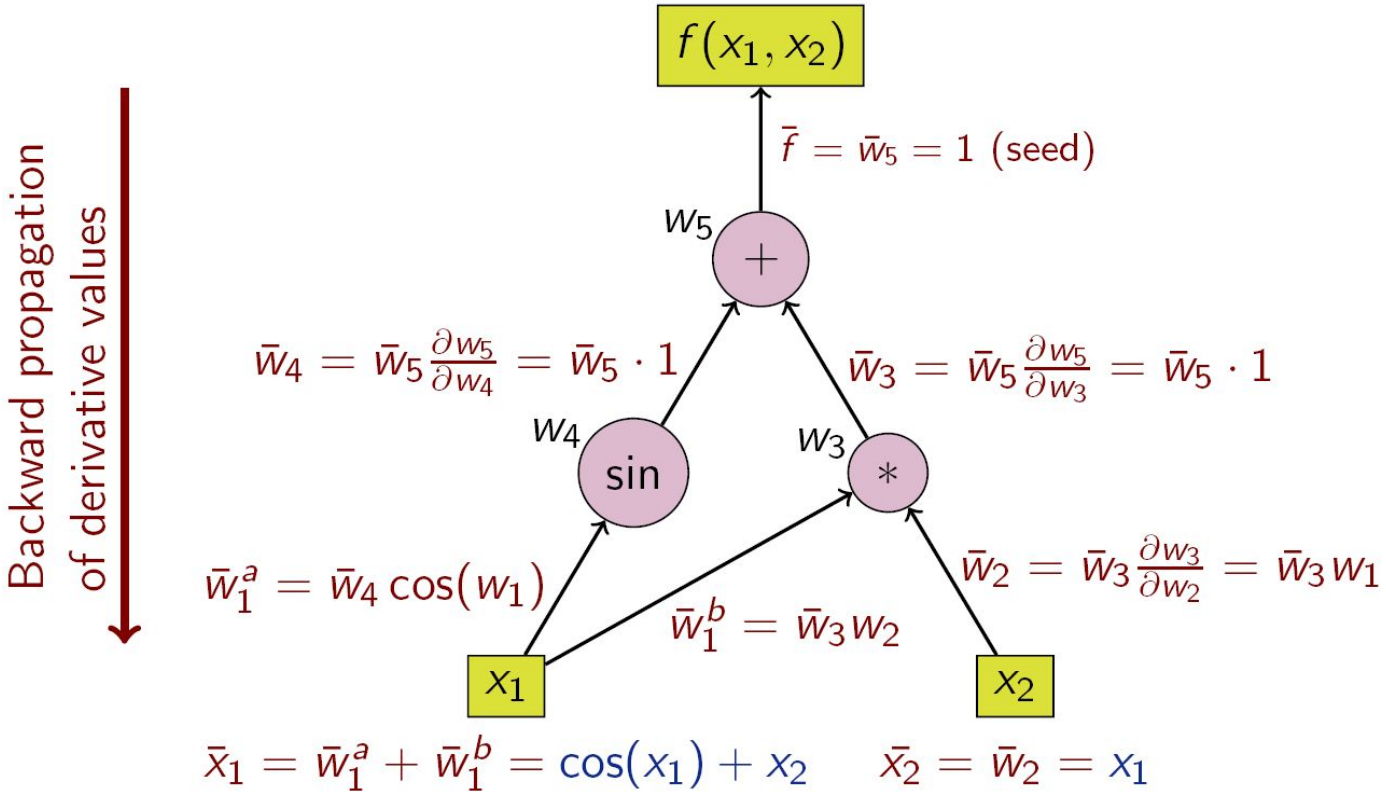
Часть 1. Производные на графе

Подсчет градиента для произвольной архитектуры

Хотим иметь возможность для произвольной сети уметь считать производную градиента по весам. Три варианта:

- Численные производные f(w + dw) - f(w) / dw
- Символьное дифференцирование (математические пакеты)
- **Автоматическое дифференцирование** (производные на любых направленных ациклличных графах DAG)

Производная на графе



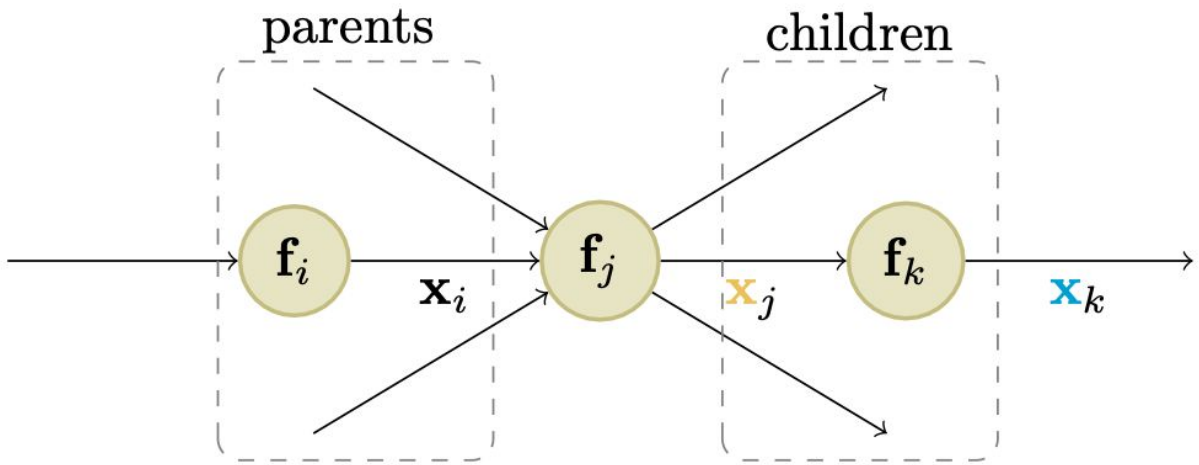
$$
\begin{aligned}
z &= f(x_1, x_2) \\
&= x_1 x_2 + \sin x_1 \\
&= w_1 w_2 + \sin w_1 \\
&= w_3 + w_4 \\
&= w_5
\end{aligned}
$$

$$
\frac{\partial o}{\partial x_j} = \sum_{k \in \mathrm{Ch}(j)} \frac{\partial o}{\partial x_k} \frac{\partial x_k}{\partial x_j}
$$

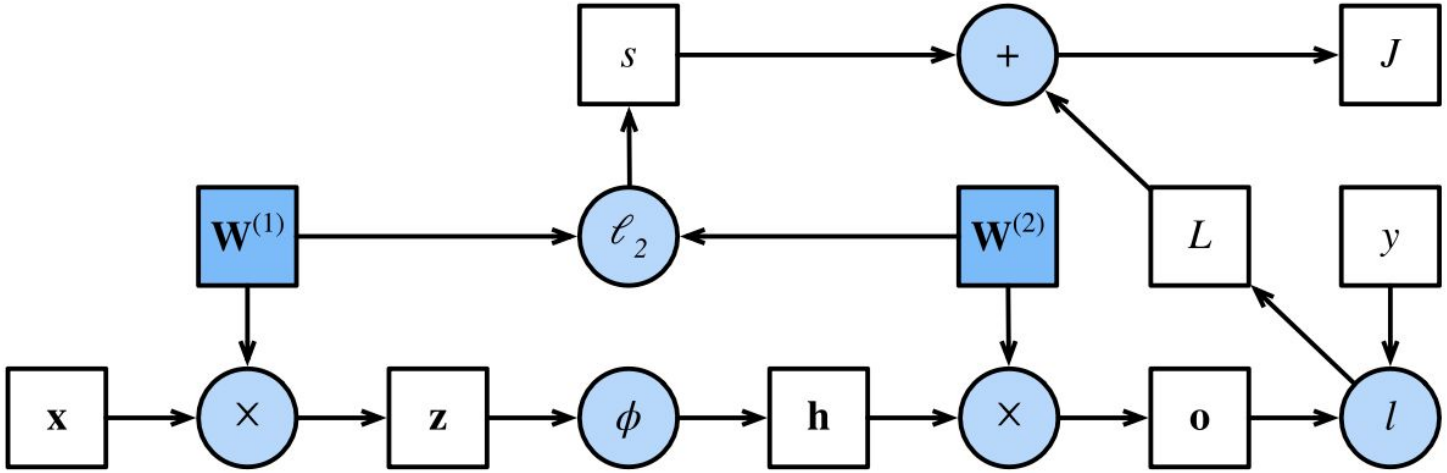
Граф ацикличен, обходим в порядке топологической сортировки

Сначала считаем выход, **запоминаем все, что необходимо**

Производная на графе



Источник:
probml.github.io/pml-book/book1.html



Два скрытых слоя, l2 регуляризация, лосс l + s

Forward, backward

Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network.
Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.
`grad_table`$[u^{(n)}] \leftarrow 1$
for $j = n - 1$ down to 1 **do**
  The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:
  `grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$
end for
return {`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i$}

Источник:
deeplearningbook.org

Пример

150 строчек кода на Питоне
github.com/karpathy/micrograd
Ничего не векторизовано, поэтому в реальной работе так делать нельзя. Можно понять принцип, современных библиотек для автоматического дифференцирования.

Часть 2. Бэкпроп для полносвязной сети

Полносвязная сеть

$\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$ g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ f maps from $\mathbb{R}^n$ to $\mathbb{R}$. $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$,

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}.$$

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^\top \nabla_{\boldsymbol{y}} z,$$

$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

Все что нужно сделать, это умножать Якобиан на вектор выходных градиентов!

Источник: deeplearningbook.org

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^{\mathrm{T}} f_1 \\ \vdots \\ \nabla^{\mathrm{T}} f_m \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Произведение вектора на Якобиан для некоторых слоев

$$z = f(\boldsymbol{x}) = \text{CrossEntropyWithLogits}(\boldsymbol{y}, \boldsymbol{x}) = -\sum_c y_c \log(\text{softmax}(\boldsymbol{x})_c) = -\sum_c y_c \log p_c$$

$$z = f(\boldsymbol{x}) = -\log(p_c) = -\log\left(\frac{e^{x_c}}{\sum_j e^{x_j}}\right) = \log\left(\sum_j e^{x_j}\right) - x_c$$

$$\frac{\partial z}{\partial x_i} = \frac{\partial}{\partial x_i}\log\sum_j e^{x_j} - \frac{\partial}{\partial x_i}x_c = \frac{e^{x_i}}{\sum_j e^{x_j}} - \frac{\partial}{\partial x_i}x_c = p_i - \mathbb{I}(i = c)$$

$$\mathbf{J} = \frac{\partial z}{\partial \boldsymbol{x}} = (\boldsymbol{p} - \boldsymbol{y})^\intercal \in \mathbb{R}^{1 \times C}$$

Заметьте, что Якобиан это вектор-строка, так как выход скалярный!

Источник: probml.github.io/pml-book/book1.html

Произведение вектора на Якобиан для некоторых слоев

Любая нелинейность

$$\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{x}) = \varphi(\boldsymbol{x}), \text{ so } z_i = \varphi(x_i).$$

$$\frac{\partial z_i}{\partial x_j} = \begin{cases} \varphi'(x_i) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbf{J} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \operatorname{diag}(\varphi'(\boldsymbol{x}))$$

$$\varphi(a) = \operatorname{ReLU}(a) = \max(a, 0)$$

$$\varphi'(a) = \begin{cases} 0 & a < 0 \\ 1 & a > 0 \end{cases}$$

Якобиан это диагональная матрица, поэтому при умножении на вектор надо просто поэлементно умножить на диагональные элементы!

Источник:
probml.github.io/pml-book/book1.html

Произведение вектора на Якобиан для некоторых слоев

Линейный слой

Источник: probml.github.io/pml-book/book1.html

$$\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{x}, \mathbf{W}) = \mathbf{W}\boldsymbol{x}, \quad \mathbf{W} \in \mathbb{R}^{m \times n}, \quad \boldsymbol{x} \in \mathbb{R}^n \quad \boldsymbol{z} \in \mathbb{R}^m \qquad z_i = \sum_{k=1}^{n} W_{ik} x_k \qquad \mathbf{J} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \in \mathbb{R}^{m \times n}$$

$$\frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} \sum_{k=1}^{n} W_{ik} x_k = \sum_{k=1}^{n} W_{ik} \frac{\partial}{\partial x_j} x_k = W_{ij} \qquad \mathbf{J} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \mathbf{W}$$

Якобиан по входу

$$z_k = \sum_{l=1} W_{kl} x_l$$

$$\mathbf{J} = \frac{\partial \boldsymbol{z}}{\partial \mathbf{W}} \quad m \times (m \times n)$$

$$\frac{\partial z_k}{\partial W_{ij}} = \sum_{l=1}^{m} x_l \frac{\partial}{\partial W_{ij}} W_{kl} = \sum_{l=1}^{m} x_l \mathbb{I}\left(i = k \text{ and } j = l\right)$$

$$\frac{\partial \boldsymbol{z}}{\partial W_{ij}} = \begin{pmatrix} 0 & \cdots & 0 & x_j & 0 & \cdots & 0 \end{pmatrix}^{\mathsf{T}}$$

$$\left[\boldsymbol{u}^{\mathsf{T}} \frac{\partial \boldsymbol{z}}{\partial \mathbf{W}}\right]_{1,:} = \boldsymbol{u}\boldsymbol{x}^{\mathsf{T}}$$

$$\boldsymbol{u}^{\mathsf{T}} \frac{\partial \boldsymbol{z}}{\partial W_{ij}} = \sum_{k=1}^{m} u_k \frac{\partial z_k}{\partial W_{ij}} = u_i x_j$$

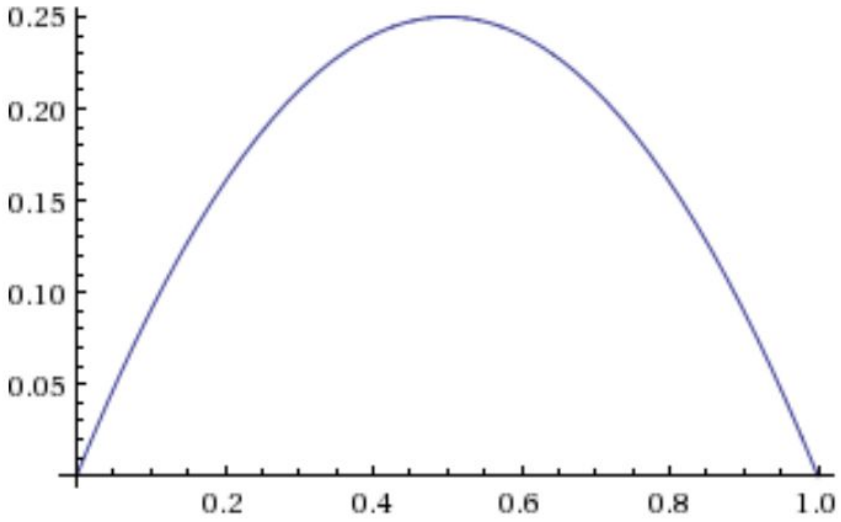
Якобиан на вектор по весам
Теперь можем обучить фулл коннект сеть

Производная сигмоида

Рассмотрим в качестве функции активации логистическую функцию:

$$\begin{aligned} \sigma(z) &= \frac{1}{1+e^{-z}} \\ \frac{\partial \sigma(z)}{\partial z} &= \sigma(z) \cdot (1 - \sigma(z)) \end{aligned}$$

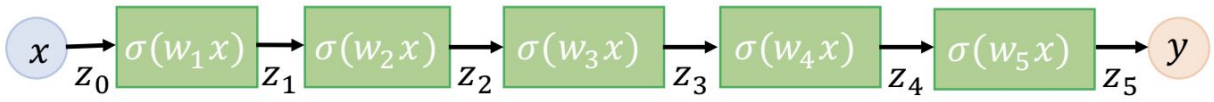
Построим график значений производной:



- максимум равен $\sigma_{\max} = \frac{1}{4}$

Затухание/взрыв градиента

Рассмотрим простую сеть (один нейрон в каждом слое):



Прямой проход:

$x = z_0$

$z_k = \sigma(z_{k-1} w_k)$

$y = z_5$

Вычислим градиенты весов для $L(y,t) = \frac{1}{2}(y_j - t_j)^2$:

$$\begin{aligned}
\frac{\partial L}{\partial z_4} &= \frac{\partial L}{\partial y}\frac{\partial y}{\partial z_4} = \overbrace{(y-t)}^{\leq 2}\overbrace{\sigma'(w_5 z_4)}^{\leq \frac{1}{4}} w_5 \leq 2 \cdot \frac{1}{4} w_5 \\
\frac{\partial L}{\partial z_3} &= \frac{\partial L}{\partial z_4}\frac{\partial z_4}{\partial z_3} \leq 2 \cdot (\frac{1}{4})^2 w_4 w_5 \\
\frac{\partial L}{\partial x} &= \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial x} \leq 2 \cdot (\frac{1}{4})^5 w_1 w_2 w_3 w_4 w_5
\end{aligned}$$

Theorem. Let $A \in \mathbf{C}^{n\times n}$ with spectral radius $\rho(A)$. Then $\rho(A) < 1$ if and only if

$$\lim_{k\to\infty} A^k = 0.$$

On the other hand, if $\rho(A) > 1$, $\lim_{k\to\infty} \|A^k\| = \infty$. The statement holds for any choice of matrix norm on $\mathbf{C}^{n\times n}$.

Для многомерного случая надо смотреть на спектральный радиус матрицы W

Паралич в сети

input [841]	layer -5	layer -4	layer -3	layer -2	layer -1	output
neurons	100	100	100	100	100	26
|grad|	6.2e-8	2.2e-6	1.6e-5	1.1e-4	7e-4	0.015

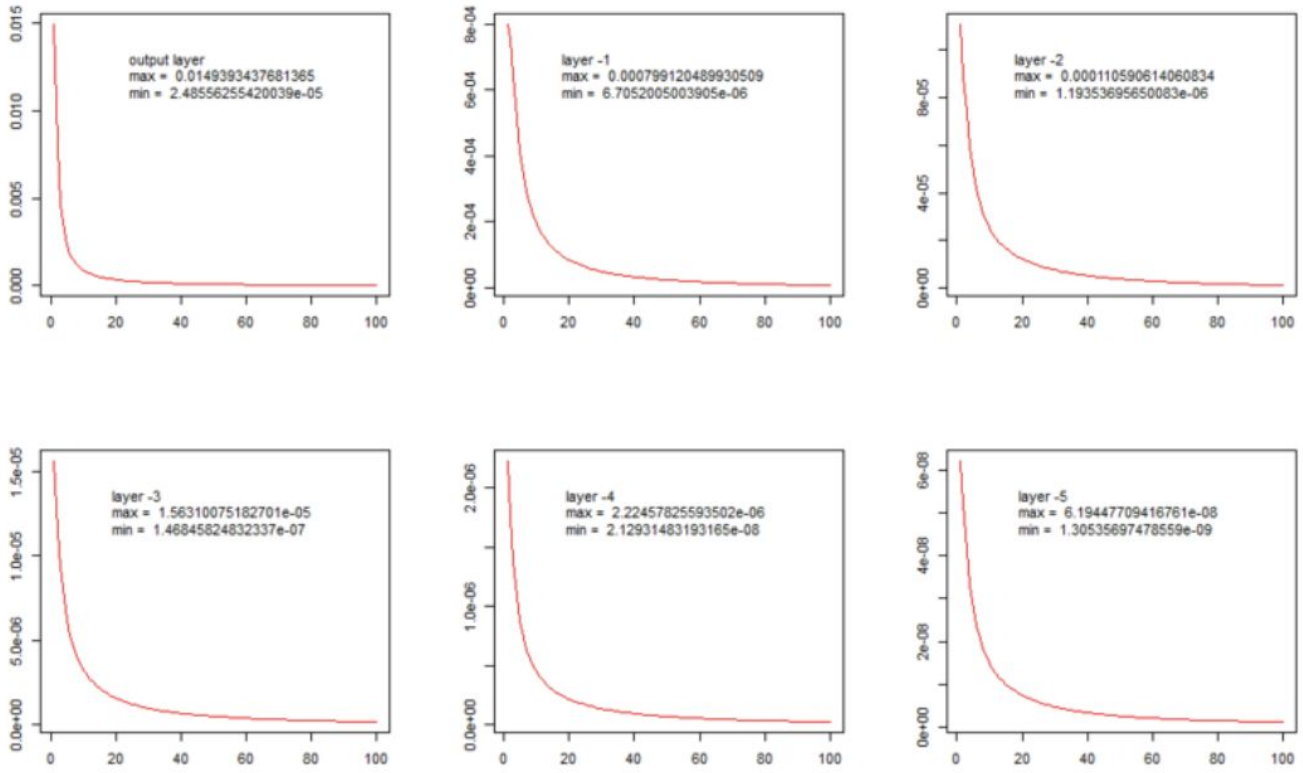
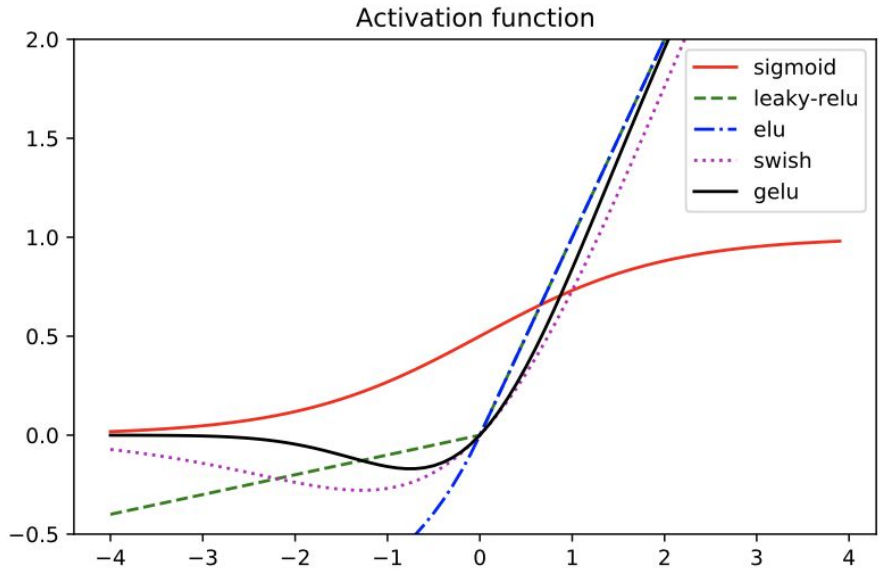


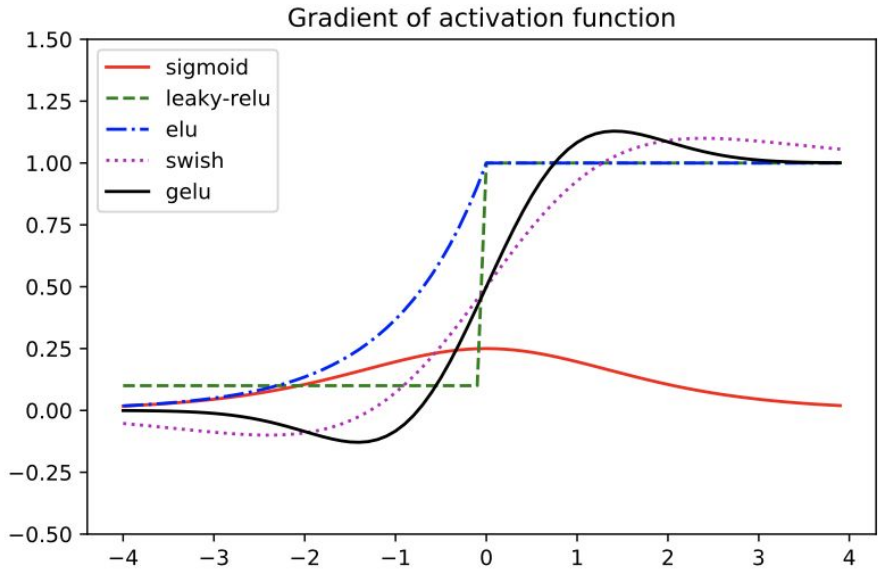
Figure: Средний модуль градиента в различных слоях

Функции активации

Sigmoid	$\sigma(a) = \frac{1}{1+e^{-a}}$
Hyperbolic tangent	$\tanh(a) = 2\sigma(2a) - 1$
Softplus	$\sigma_+(a) = \log(1 + e^a)$
Rectified linear unit	$\mathrm{ReLU}(a) = \max(a, 0)$
Leaky ReLU	$\max(a, 0) + \alpha \min(a, 0)$
Exponential linear unit	$\max(a, 0) + \min(\alpha(e^a - 1), 0)$
Swish	$a\sigma(a)$
GELU	$a\Phi(a)$

Источник:
probml.github.io/pml-book/book1.html





Регуляризация

Дополнительный штраф: $L_R = L\left(\vec{y}, \vec{t}\right) + \lambda \cdot R(W)$

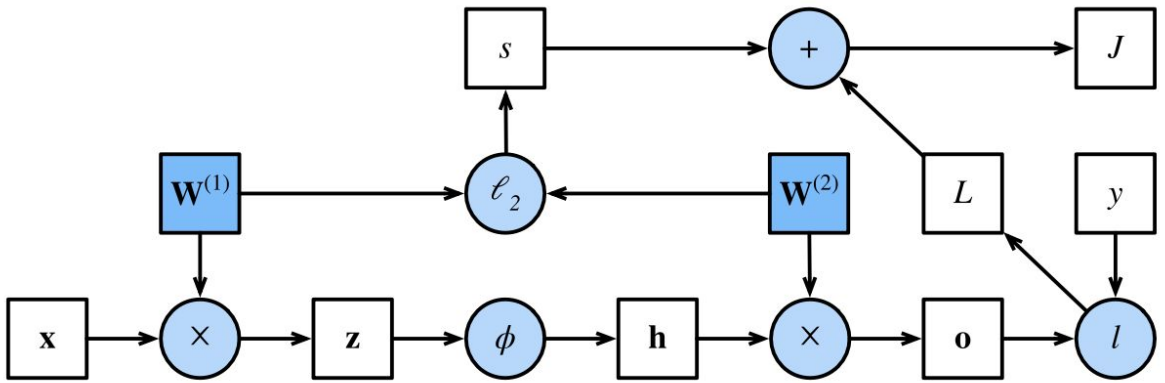
L2 регуляризация:

- $R_{L2}(W) = \frac{1}{2}\sum_i w_i^2$
- $\frac{\partial R_{L2}(W)}{\partial w_i} = w_i$
- Помогает бороться с мультиколлинеарностью

L1 регуляризация:

- $R_{L1}(W) = \sum_i |w_i|$
- $\frac{\partial R_{L1}(W)}{\partial w_i} = sign(w_i)$
- Поощряет разреженные веса

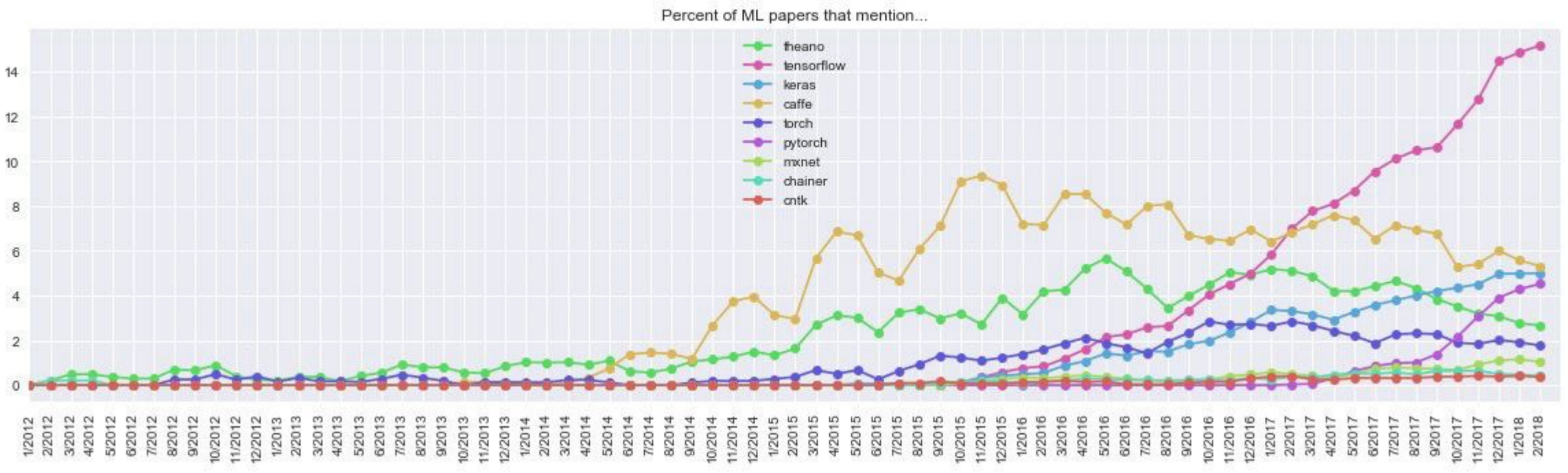
Часто называется **weight decay**



Источник:
probml.github.io/pml-book/book1.html

Часть 3. Pytorch

Популярность фреймворков



Популярность по использованию в публикациях

Источник medium.com/neuromation-blog

На Pytorch **очень удобно писать**, как будто пишешь на Питоне

Tensor

```
import torch

x = torch.arange(4.0)
x
```

Создаем тензор

```
tensor([0., 1., 2., 3.])
```

```
x.requires_grad_(True)  # Better create `x = torch.arange(4.0, requires_grad=True)`
x.grad                  # The default value is None
```

Хотим считать градиент

```
y = 2 * torch.dot(x, x)
y
```

```
tensor(28., grad_fn=<MulBackward0>)
```

Источник d2l.ai

```
y.backward()
x.grad
```

Градиент dy / dx

```
tensor([ 0.,  4.,  8., 12.])
```

Tensor

```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))  # Faster: y.sum().backward()
x.grad
```

```
tensor([0., 2., 4., 6.])
```

```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

z.sum().backward()
x.grad == u
```

Убираем переменную
из графа

```
tensor([True, True, True, True])
```

```
def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

Источник d2l.ai

Module

```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens) * sigma)
        self.b1 = nn.Parameter(torch.zeros(num_hiddens))
        self.W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * sigma)
        self.b2 = nn.Parameter(torch.zeros(num_outputs))
```

```
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2
```

Можно делать модель напрямую с помощью тензоров

Главное реализовать метод forward

Источник d2l.ai

А можно использовать готовые модули!

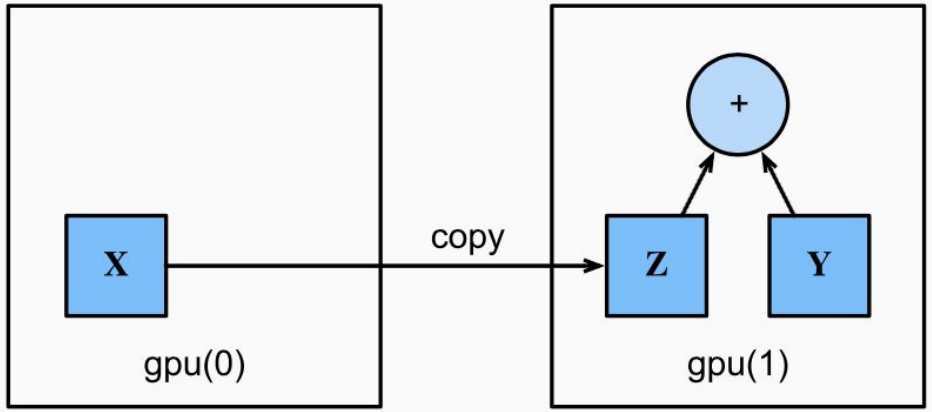
```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

Собственные слои

```python
class LinearFunction(Function):

    # Note that both forward and backward are @staticmethods
    @staticmethod
    # bias is an optional argument
    def forward(ctx, input, weight, bias=None):
        ctx.save_for_backward(input, weight, bias)
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    # This function has only a single output, so it gets only one gradient
    @staticmethod
    def backward(ctx, grad_output):
        # This is a pattern that is very convenient - at the top of backward
        # unpack saved_tensors and initialize all gradients w.r.t. inputs to
        # None. Thanks to the fact that additional trailing Nones are
        # ignored, the return statement is simple even when the function has
        # optional inputs.
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None

        # These needs_input_grad checks are optional and there only to
        # improve efficiency. If you want to make your code simpler, you can
        # skip them. Returning gradients for inputs that don't require it is
        # not an error.
        if ctx.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if ctx.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and ctx.needs_input_grad[2]:
            grad_bias = grad_output.sum(0)

        return grad_input, grad_weight, grad_bias
```

```python
class Linear(nn.Module):
    def __init__(self, input_features, output_features, bias=True):
        super(Linear, self).__init__()
        self.input_features = input_features
        self.output_features = output_features

        # nn.Parameter is a special kind of Tensor, that will get
        # automatically registered as Module's parameter once it's assigned
        # as an attribute. Parameters and buffers need to be registered, or
        # they won't appear in .parameters() (doesn't apply to buffers), and
        # won't be converted when e.g. .cuda() is called. You can use
        # .register_buffer() to register buffers.
        # nn.Parameters require gradients by default.
        self.weight = nn.Parameter(torch.empty(output_features, input_features))
        if bias:
            self.bias = nn.Parameter(torch.empty(output_features))
        else:
            # You should always register all possible parameters, but the
            # optional ones can be None if you want.
            self.register_parameter('bias', None)

        # Not a very smart way to initialize weights
        nn.init.uniform_(self.weight, -0.1, 0.1)
        if self.bias is not None:
            nn.init.uniform_(self.bias, -0.1, 0.1)

    def forward(self, input):
        # See the autograd section for explanation of what happens here.
        return LinearFunction.apply(input, self.weight, self.bias)
```

GPU




```
def gpu(i=0):  #@save
    return torch.device(f'cuda:{i}')
```

Источник d2l.ai

```
x = torch.tensor([1, 2, 3])
x.device
```

```
device(type='cpu')
```

```
net = nn.Sequential(nn.LazyLinear(1))
```

net.cuda()

Dataset

```python
class FaceLandmarksDataset(Dataset):
    """Face Landmarks dataset."""

    def __init__(self, csv_file, root_dir, transform=None):
        """
        Args:
            csv_file (string): Path to the csv file with annotations.
            root_dir (string): Directory with all the images.
            transform (callable, optional): Optional transform to be applied
                on a sample.
        """
        self.landmarks_frame = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.landmarks_frame)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        img_name = os.path.join(self.root_dir,
                                self.landmarks_frame.iloc[idx, 0])
        image = io.imread(img_name)
        landmarks = self.landmarks_frame.iloc[idx, 1:]
        landmarks = np.array([landmarks])
        landmarks = landmarks.astype('float').reshape(-1, 2)
        sample = {'image': image, 'landmarks': landmarks}

        if self.transform:
            sample = self.transform(sample)

        return sample
```

- map-style dataset – get_item
- iterable-style dataset – iter

Dataloader

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
           batch_sampler=None, num_workers=0, collate_fn=None,
           pin_memory=False, drop_last=False, timeout=0,
           worker_init_fn=None, *, prefetch_factor=2,
           persistent_workers=False)
```

```
def collate_batch(batch):

  label_list, text_list, = [], []

  for (text,label) in batch:
    label_list.append(label)
    clear_text = torch.tensor(text_pipeline(text), dtype=torch.int64)
    text_list.append(clear_text)

  label_list = torch.tensor(label_list, dtype=torch.int64)

  text_list = pad_sequence(text_list, padding_value=0)

  return text_list, label_list,
```

```
for i_batch, sample_batched in enumerate(dataloader):
```

Train-test loop

Обучение

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0
```

Тестирование

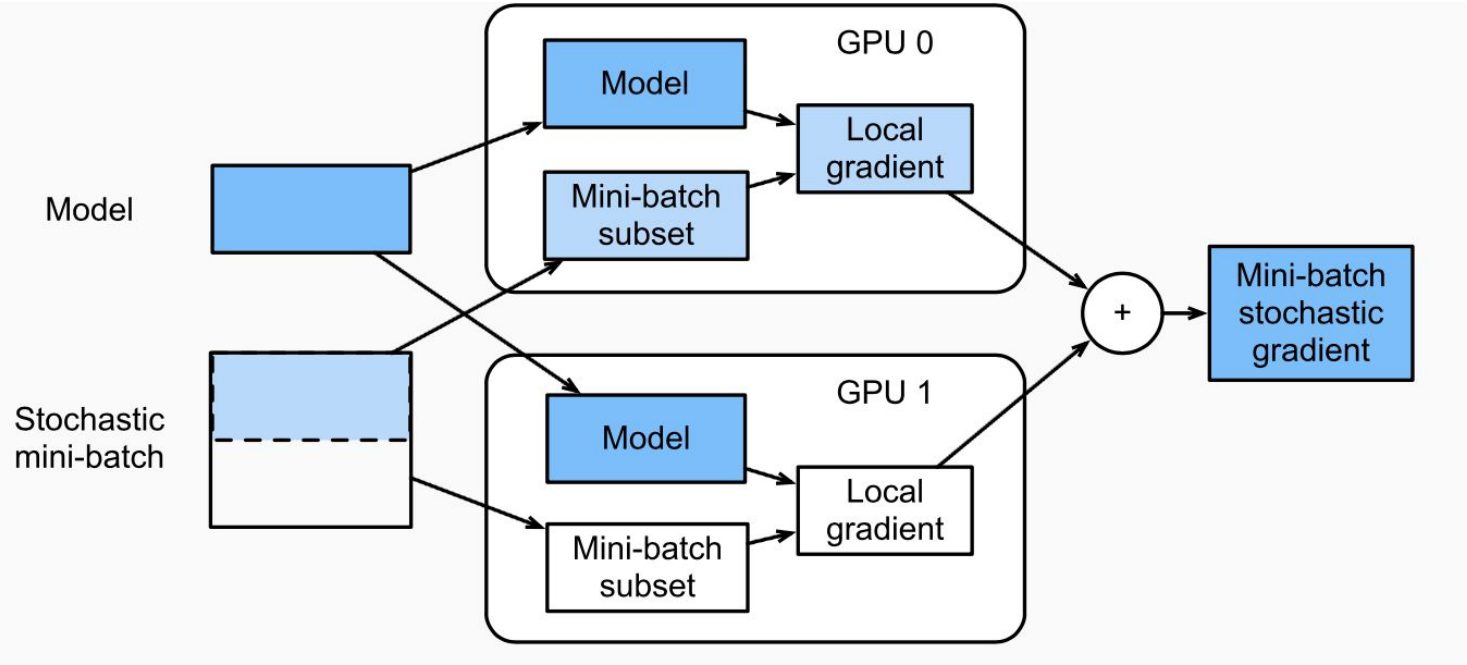
```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')
```

Distributed learning

`torch.nn.DataParallel – много гпу на одной машине`

`torch.nn.parallel.DistributedDataParallel – много гпу на разных машинах`

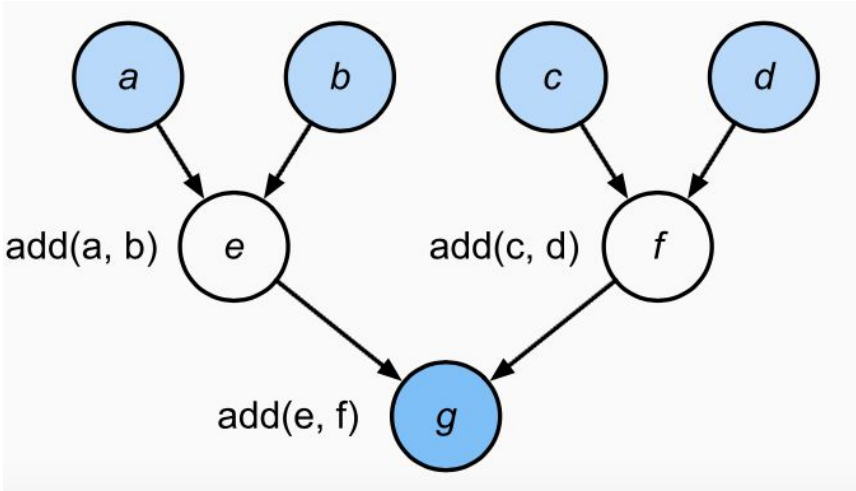
```
net = nn.DataParallel(net)
l = loss(net(X), y)
l.backward()
```



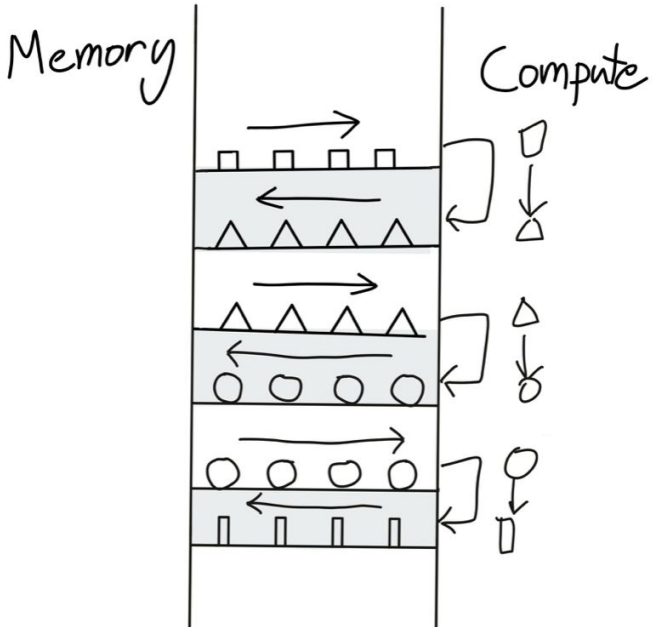
Источник d2l.ai

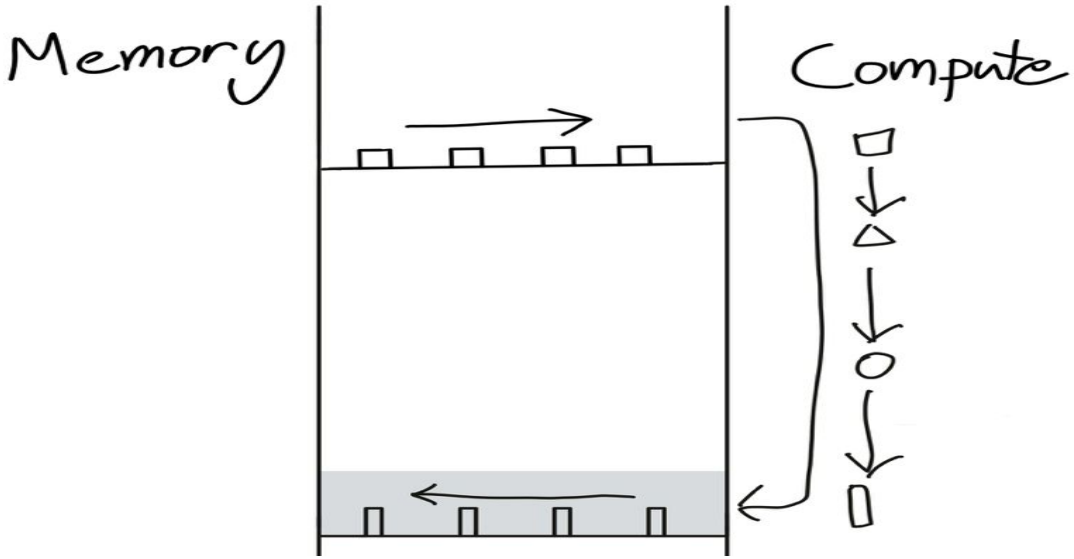
Fusion операций

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
```



Источник d2l.ai





Источник
horace.io/brrr_intro.html

JIT

Позволяет значительно ускорить код, особенно там, где много работает python.

Сериализация Pytorch модели в standalone C++ программу.

```
net = get_net()
with Benchmark('Without torchscript'):
    for i in range(1000): net(x)

net = torch.jit.script(net)
with Benchmark('With torchscript'):
    for i in range(1000): net(x)
```

Источник d2l.ai

Слайд для вопросов

Вопросы?