Adam

Momentum

Stochastic gradient descent momentum introduces a momentum term that accumulates the past gradient. This helps speed up the convergence, especially in the presence of some noisy or sparse gradient.

Set v_t as the momentum term (velocity) of the iteration t and β as the momentum coefficient. The update rule is like this:

$$v_{t+1} = \beta \cdot v_t + (1 - \beta) \cdot \nabla J(\theta_t)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot v_{t+1}$$

RMSprop

RMSprop, which stands for root mean squared propagation, is designed to address some limitations of traditional SGD algorithm. RMSprop maintains a running average of the squared gradients.

The learning rates for each parameters are then normalized by dividing them by the squared root of the running average. The normalization helps to prevent the gradient from too small or too large.

The update rule of RMSprop goes like this:

$$E[g^2]_t = \beta \cdot E[g^2]_t + (1 - \beta) \cdot \nabla J(\theta_t)^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} \cdot \nabla J(\theta_t)$$

Adam

Adam, also known as Adaptive Moment Estimation, is the combination of Momentum and RMSprop. That is to say, we compute v_t and $E[g^2]_t$ during the iterations and modify the formula like this:

$$v_{t+1} = \beta_1 \cdot v_t + (1 - \beta_1) \cdot \nabla J(\theta_t)$$

$$E[g^2]_t = \beta_2 \cdot E[g^2]_t + (1 - \beta_2) \cdot \nabla J(\theta_t)^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} \cdot v_t$$

In practice, we often set $\beta_1 = 0.9, \beta_2 = 0.99$. It turns out the configuration is optimal in the most cases.

Since the moving averages are initialized with zeros, they are biased towards zero, especially during the early iterations. To address this bias, Adam incorporates a **bias correction** mechanism. Additionally, compute:

$$v'_t = \frac{v_t}{1 - \beta_1^t}$$

$$E[g^2]'_t = \frac{E[g^2]'_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]'_t + \epsilon}} \cdot v'_t$$

Check the code to get a better understanding of the Adam algorithm.

```
keys = ["learning_rate", "beta1", "beta2",
"epsilon", "m", "v", "t"]
learning_rate, beta1, beta2, epsilon, m, v, t =
[config.get(key) for key in keys]
t += 1 # start at 1
m = beta1 * m + (1 - beta1) * dw # momentum
v = beta2 * v + (1 - beta2) * (dw**2) # RMSprop
m_unbia = m / (1 - beta1 ** t) # correct the bia
in the early stage
v_unbia = v / (1 - beta2 ** t) # correct the bia
in the early stage
next_w = w - learning_rate * m_unbia /
np.sqrt(v_unbia + epsilon)
config["m"], config["v"], config["t"] = m, v, t
```

Batch Normalization

[Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (arxiv.org)]

Intuition

To understand the goal of batch normalization, it is important to recognize that machine learning methods tend to perform better with input data consisting of uncorrelated feature with zero mean and unit variance.

When training a neural network, we preprocess the data before feeding it into the network. This will ensure that the data in the first layer follows a nice distribution. However, the activation functions in the deep layer will break the nice property.

The authors of batch normalization hypothesize that the shifting distribution of features inside deep neural networks may make training deep network more difficult. To overcome the problem, they propose to insert a layer that normalizes batches.

At training time, such a layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These parameters are then used to center and normalize the features. A running average of the mean and standard deviation is kept during the training process, which will be applied to the test data.

Implementation

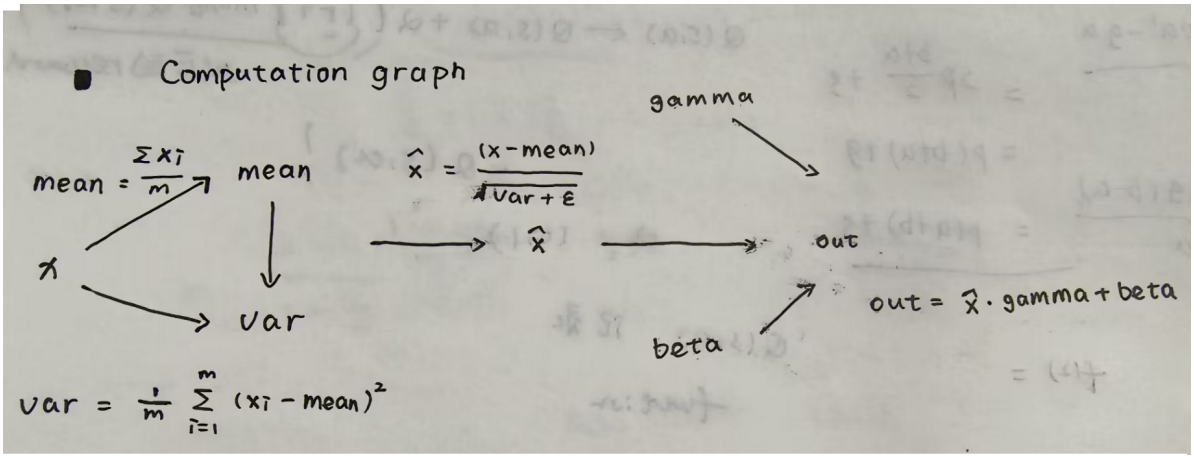
It is possible that this normalization strategy could reduce the representational power of the network, since it may sometimes be optimal for certain layers to have features that are not zero-mean or unit variance.

To this end, the batch normalization layer includes learnable shift and scale parameters for each feature dimension. This is the implementation of the batch normalization:

```
if mode == "train":
    mean = x.mean(axis = 0) # (D, )
    var = x.var(axis = 0) # (D, )
    x_hat = (x - mean) / np.sqrt(var + eps) # zero
mean and one unit
    out = x_hat * gamma + beta # the target scale
and shift
    # momentum = 0.9 (it is a common way to
compute the running average)
    running_mean = momentum * running_mean + (1 -
momentum) * mean
    running_var = momentum * running_var + (1 -
momentum) * var
elif mode == "test":
    # use the running average to normalize the
test data
    # the running average is fixed now, so the
distribution will be fixed.
    x_hat = (x - running_mean) /
np.sqrt(running_var + eps)
    out = x_hat * gamma + beta
```

Backpropagation

To derive the backward pass you should write out the computation graph for batch normalization and backprop through each of the intermediate nodes. Some intermediates may have multiple outgoing branches; make sure to sum gradients across these branches in the backward pass.



Tips: compute the gradient according to the topological order.

$$\frac{\partial L}{\partial \hat{x}_i} = \frac{\partial L}{\partial y_i} \cdot \gamma$$

$$\frac{\partial L}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot (x_i - \mu) \cdot \frac{-1}{2}(\sigma^2 + \epsilon)^{-3/2}$$

$$\frac{\partial L}{\partial \mu} = \left(\sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma^2+\epsilon}}\right) + \frac{\partial L}{\partial \sigma^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i-\mu)}{m}$$

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma^2+\epsilon}} + \frac{\partial L}{\partial \sigma^2} \cdot \frac{2(x_i-\mu)}{m} + \frac{1}{m} \cdot \frac{\partial L}{\partial \mu}$$

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \cdot \hat{x}_i$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}$$

```
m = x.shape[0]
dhat = dout * gamma
dvar = (dhat * (x - mean) * (-0.5) * ((var + eps)
** (-1.5))).sum(axis = 0)
dmean = (dhat * -((var + eps) ** (-0.5)) + dvar *
-2 * (x - mean) / m).sum(axis = 0)
dx = dhat / ((var + eps) ** 0.5) + dvar * 2 * (x-
mean) / m + dmean / m
dgamma = (dout * x_hat).sum(axis = 0)
dbeta = dout.sum(axis = 0)
```

Layer normalization

Batch normalization has proved to be effective in making networks easier to train, but the dependency on batch size makes it less useful in complex networks.

Instead of normalizing over the batch, we normalize over the features. In other words, when using Layer Normalization, each feature vector corresponding to a single datapoint is normalized based on the sum of all terms within that feature vector.

Actually, if we apply batch normalization to the transpose of matrix A. That is as same as layer normalization. However, in the layer normalization, we compute the mean and variance from script rather than use the running average in the test process.

Dropout

Dropout is a technique for regularizing neural networks by randomly setting some output activations to zero during the forward pass.

There are two advantages of dropout. Firstly, dropout introduce some kind of stochasticity in the training time and average them out in the test time (recall **model ensembles**). What's more, dropout prevents co-adaptations, where a feature detector is only helpful in the context of several other feature detectors.

Determine a constant p as the probability of setting each value to zero. To ensure the expected value remains the same after the dropout, we **divide** each values by p.

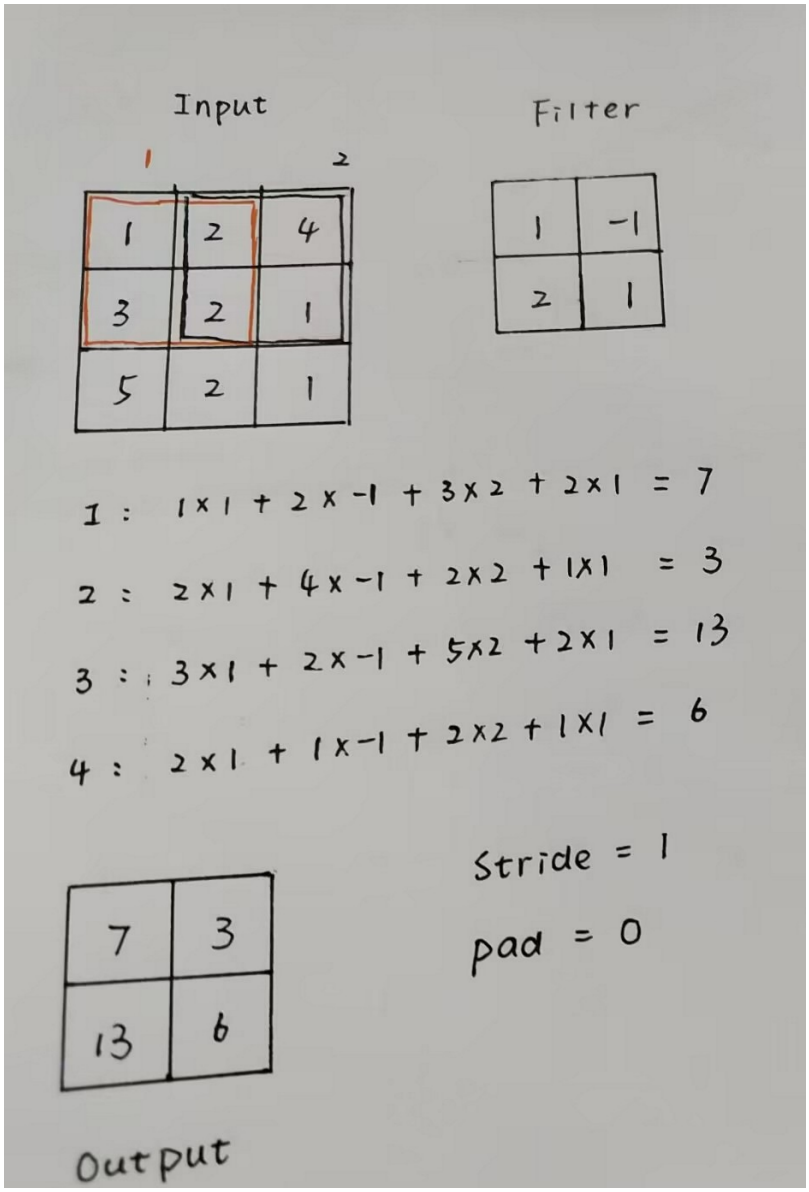
```
def dropout_forward(x, dropout_param):
```

```
    p, mode = dropout_param["p"],
dropout_param["mode"]
    if "seed" in dropout_param:
        np.random.seed(dropout_param["seed"])
    mask = None
    out = None
    if mode == "train":
        mask = (np.random.rand(*x.shape) < p) / p
        out = mask * x
    elif mode == "test":
        out = x
    cache = (dropout_param, mask)
    out = out.astype(x.dtype, copy=False)
    return out, cache
```

Convolutional Layer

Convolutional layers are a fundamental building block of Convolutional Neural Network (CNN). These layers apply convolutional operation to the input data.

Each convolutional layer has its filter. Convolution operation involves computing the dot product of the filter and a small region of the input data, and then sliding the filter across the entire input. This operation captures the features of the input data. Specifically, the process of CNN goes like this:



In practice, there are some additional concepts of convolutional layer:

- Strides: Strides determine the step size with which the filter slides over the input (Default is one).
- Padding: Padding involves adding extra pixels around the input data, which helps preserve spatial information and prevent the output size from shrinking too much.
- Channel: Channel refers to the different layers of information within an image. An image typically consists of three color channels(red, green, and blue).

Check the code to gain a better intuition:

```
def conv_forward_naive(x, w, b, conv_param):
    """
    N : number of input data
    C : number of channels
    H,w : the scale of the input image
    Input:
    - x: Input data of shape (N, C, H, W)
    - w: Filter weights of shape (F, C, HH, WW)
    - b: Biases, of shape (F,)
    - conv_param: A dictionary with the following
keys:
      - 'stride': The number of pixels between
adjacent receptive fields in the
        horizontal and vertical directions.
      - 'pad': The number of pixels that will be
used to zero-pad the input.
    Returns a tuple of:
    - out: Output data, of shape (N, F, H', W')
where H' and W' are given by
      H' = 1 + (H + 2 * pad - HH) / stride
      W' = 1 + (W + 2 * pad - WW) / stride
    - cache: (x, w, b, conv_param)
    """
    # pay attention to the SHAPE will help you to
understand the algorithm
    stride, pad = conv_param["stride"],
conv_param["pad"]
    N, C, H, W = x.shape
    F, C, HH, WW = w.shape
    # padding
    H_pad = H + 2 * pad
    W_pad = W + 2 * pad
    x_pad = np.zeros(shape = (N, C, H_pad, W_pad),
dtype = np.float64)
```

```
    for i in range(N):
        for j in range(C):
            x_pad[i, j, pad: -pad, pad: -pad] =
x[i, j]
    # stride
    H_out = 1 + int((H_pad - HH) / stride)
    W_out = 1 + int((W_pad - WW) / stride)
    out = np.zeros(shape = (N, F, H_out, W_out),
dtype = np.float64)
    for i in range(N):
        for j in range(F):
            for u in range(H_out):
                for v in range(W_out):
                    # compute the dot product of
two matrix
                    out[i, j, u, v] = np.sum(
                        x_pad[i, :, u*stride:
u*stride+HH, v*stride: v*stride+WW] * w[j]
                      ) + b[j]
    cache = (x, x_pad, w, b, conv_param)
    return out, cache
```