Regularization

L2

Regularization

- Adding regularization to NN will help it reduce variance (overfitting)
- L1 matrix norm:
  - `||W|| = Sum(|w[i,j]|)  # sum of absolute values of all w`
- L2 matrix norm because of arcane technical math reasons is called Frobenius norm:
  - `||W||^2 = Sum(|w[i,j]|^2)      # sum of all w squared`
  - Also can be calculated as `||W||^2 = W.T * W`

L2-Train

- Regularization for NN:
  - The normal cost function that we want to minimize is: `J(W1,b1...,WL,bL) = (1/m) * Sum(L(y(i),y'(i)))`
  - The L2 regularization version: `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum((||W[l]||^2)`
  - We stack the matrix as one vector `(mn,1)` and then we apply `sqrt(w1^2 + w2^2.....)`
  - To do back propagation (old way): `dw[l] = (from back propagation)`
  - The new way: `dw[l] = (from back propagation) + lambda/m * w[l]`
  - So plugging it in weight update step:
    - ```
      w[l] = w[l] - learning_rate * dw[l]
           = w[l] - learning_rate * ((from back propagation) + lambda/m * w[l])
           = w[l] - (learning_rate*lambda/m) * w[l] - learning_rate * (from back propagation)
           = (1 - (learning_rate*lambda)/m) * w[l] - learning_rate * (from back propagation)
      ```

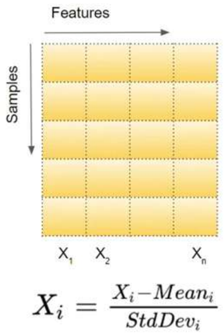
Drop-Out

```
keep_prob = 0.8   # 0 <= keep_prob <= 1
l = 3  # this code is only for layer 3
# the generated number that are less than 0.8 will be dropped. 80% stay, 20% dropped
d3 = np.random.rand(a[l].shape[0], a[l].shape[1]) < keep_prob

a3 = np.multiply(a3,d3)   # keep only the values in d3

# increase a3 to not reduce the expected value of output
# (ensures that the expected value of a3 remains the same) - to solve the scaling problem
a3 = a3 / keep_prob
```
```

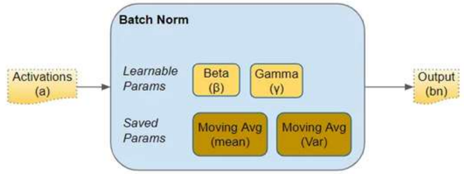
Batch Normalization



How we normalize (Image by Author)

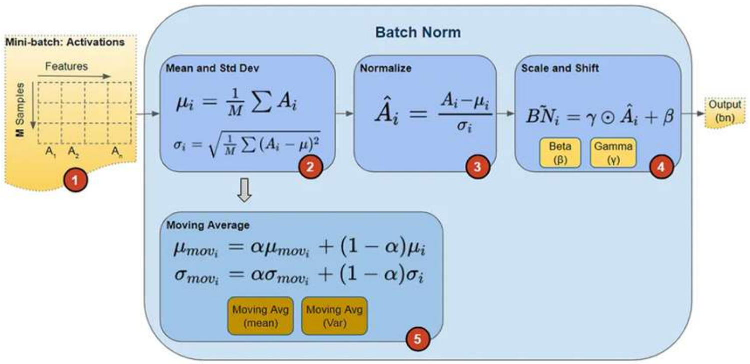
Batch Normalization

- Two learnable parameters called beta and gamma.
- Two non-learnable parameters (Mean Moving Average and Variance Moving Average) are saved as part of the 'state' of the Batch Norm layer.



Parameters of a Batch Norm layer (Image by Author)

Batch Normalization



Calculations performed by Batch Norm layer (Image by Author)

Weighted Averages

```
v = 0
Repeat
{
        Get theta(t)
        v = beta * v + (1-beta) * theta(t)
}
```

Momentum

Gradient descent with momentum

- The momentum algorithm almost always works faster than standard gradient descent.
- The simple idea is to calculate the exponentially weighted averages for your gradients and then update your weights with the new values.
- Pseudo code:

```
vdW = 0, vdb = 0
on iteration t:
        # can be mini-batch or batch gradient descent
        compute dw, db on current mini-batch

        vdW = beta * vdW + (1 - beta) * dW
        vdb = beta * vdb + (1 - beta) * db
        W = W - learning_rate * vdW
        b = b - learning_rate * vdb
```

RMS-Prop

RMSprop

- Stands for **Root mean square prop**.
- This algorithm speeds up the gradient descent.
- Pseudo code:

```
sdW = 0, sdb = 0
on iteration t:
        # can be mini-batch or batch gradient descent
        compute dw, db on current mini-batch

        sdW = (beta * sdW) + (1 - beta) * dW^2  # squaring is element-wise
        sdb = (beta * sdb) + (1 - beta) * db^2  # squaring is element-wise
        W = W - learning_rate * dW / sqrt(sdW)
        b = B - learning_rate * db / sqrt(sdb)
```

Adam optimization algorithm

- Stands for **Adaptive Moment Estimation.**
- Adam optimization and RMSprop are among the optimization algorithms that worked very well with a lot of NN architectures.
- Adam optimization simply puts RMSprop and momentum together!
- Pseudo code:

```
vdW = 0, vdW = 0
sdW = 0, sdb = 0
on iteration t:
        # can be mini-batch or batch gradient descent
        compute dw, db on current mini-batch

        vdW = (beta1 * vdW) + (1 - beta1) * dW     # momentum
        vdb = (beta1 * vdb) + (1 - beta1) * db     # momentum

        sdW = (beta2 * sdW) + (1 - beta2) * dW^2   # RMSprop
        sdb = (beta2 * sdb) + (1 - beta2) * db^2   # RMSprop

        vdW = vdW / (1 - beta1^t)      # fixing bias
        vdb = vdb / (1 - beta1^t)      # fixing bias

        sdW = sdW / (1 - beta2^t)      # fixing bias
        sdb = sdb / (1 - beta2^t)      # fixing bias

        W = W - learning_rate * vdW / (sqrt(sdW) + epsilon)
        b = B - learning_rate * vdb / (sqrt(sdb) + epsilon)
```

- Hyperparameters for Adam:
  - Learning rate: needed to be tuned.
  - `beta1` : parameter of the momentum - `0.9` is recommended by default.
  - `beta2` : parameter of the RMSprop - `0.999` is recommended by default.
  - `epsilon` : `10^-8` is recommended by default.

AdamW

Standard Adam computes L2 here

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

AdamW (Weight Decay) adds term here

```
x -= learning_rate * weight_decay * x
```

```
# 1️⃣ Compute the gradient
g_t = ∇L(W)

# 2️⃣ Update Adam's moment estimates
m_t = β1 * m_{t-1} + (1 - β1) * g_t
v_t = β2 * v_{t-1} + (1 - β2) * g_t**2

# 3️⃣ Apply bias correction
m_hat = m_t / (1 - β1**t)
v_hat = v_t / (1 - β2**t)

# 4️⃣ Update the weights (Adam update step)
W -= η * m_hat / (sqrt(v_hat) + ε)

# 5️⃣ Apply weight decay (separate from the gradient)
W -= η * λ * W
```