

Special topics: Convolutional Neural Networks

Giảng viên: TS. Nguyễn Thị Ngọc Diệp

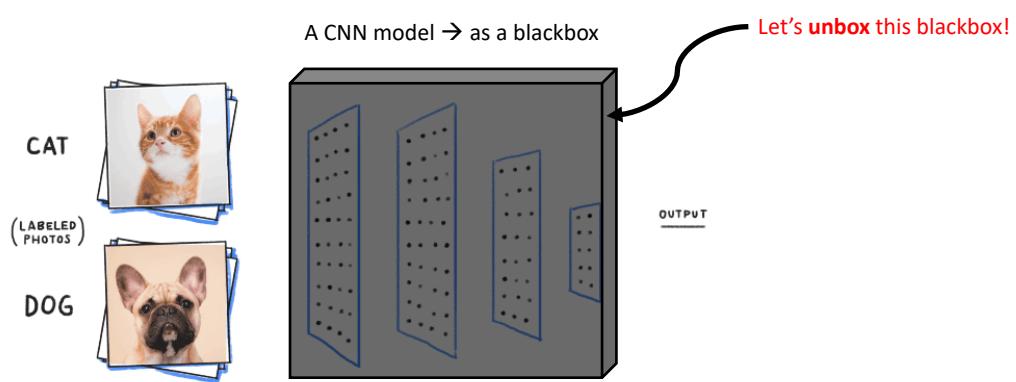
Email: ngocdiep@vnu.edu.vn

Slide & Code: https://github.com/chupibk/INT3414_22

Schedule

Week	Content	Class hour	Self-study hour
1	Introduction CNNs in Computer Vision	2	1
2	Foundations of CNNs Case study: Image classification problem Basics of Neural networks Training with backpropagation Implementation with PyTorch	2	2-6
3	Training and tuning parameters Data augmentation - Data generator Foundations of CNNs Model evaluation - Transfer learning	2	2-6
4	Object detection	2	2-6
5	Segmentation	2	2-6
6, 7	Mid-term presentations	2	2-6
8, 9	Advanced topics using CNNs	2	2-6
10, 11, 12	Final project presentations	1	2-6
13	Class summarization	1-3	open

Recall week 1



The most important: design the problem to a solvable task using the blackbox

Image credit: <https://becominghuman.ai/building-an-image-classifier-using-deep-learning-in-python-totally-from-a-beginners-perspective-be8dbaf22dd8>

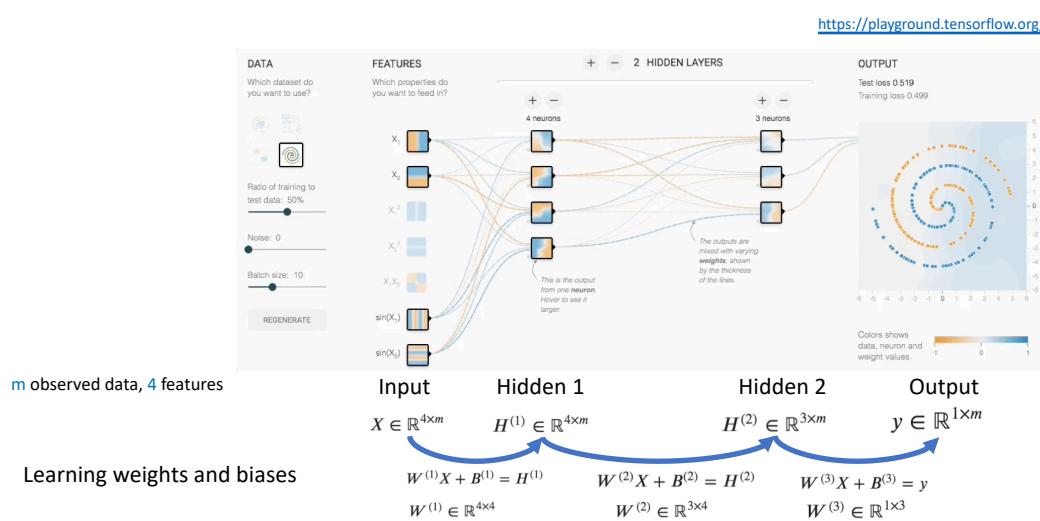
Week 2

Basics of Neural Networks and Deep Learning
Implementation with PyTorch

Code

- Google Colab notebook
 - <https://colab.research.google.com/drive/1hLRa43CbzXeE0-DY7UITrbDUeITsS7Pe>

Build a neural network



Implementation with PyTorch

```

model = nn.Sequential(
    nn.Linear(D_in, H),
    nn.ReLU(),
    nn.Linear(H, D_out),
    nn.Sigmoid()
)

loss_fn = nn.BCELoss()
lr = 0.01 #learning rate
optimizer = torch.optim.Adam(
    model.parameters(),
    lr=lr
)
for t in range(num_epochs):
    # Forward pass
    y_pred = model(x_train_t)

    # Compute loss
    loss = loss_fn(y_pred, y_train_t)

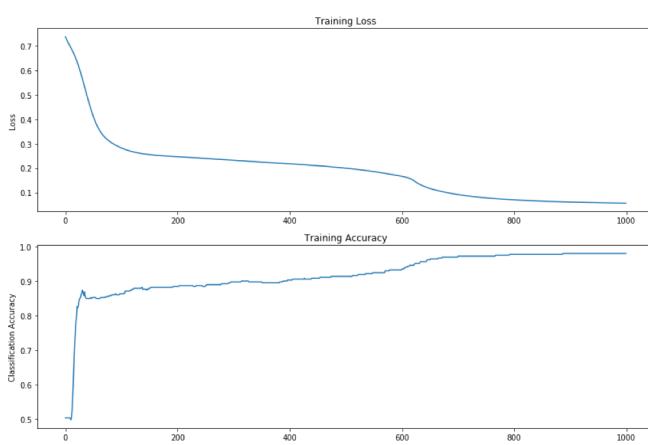
    # Clear the buffers
    optimizer.zero_grad()

    # Backward pass: compute gradients
    loss.backward()

    # update the parameters
    optimizer.step()

```

Visualize the training loss



Steps in training a CNN – Keras

- **Prepare data:** `x_train, y_train, x_val, y_val, x_test, y_test`
 - `x: (b, h, w, c)`
 - `y: (b, n)`
- **Define a CNN model**
 - Conv + dense layers
- **Compile** the model
 - Define a loss function: cross entropy loss
 - Set an optimizer & its parameters
 - Define evaluation metrics

`model.compile(...)`
- **Train** the model
 - **Feed the actual data and do weight updating**
 - Feed several times (# of epochs)
 - Feed a small amount at a time (`batch_size`)

`model.fit(...)`

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

Loss function: Cross entropy

Other form:

$p(x)$: true distribution
 $q(x)$: estimated distribution
 x : discrete variable

y : ground truth vector
 \hat{y} : estimated vector
 (\cdot) : vector dot product

$$H(p, q) = - \sum_{\forall x} p(x) \log(q(x))$$

$$L = -\mathbf{y} \cdot \log(\hat{\mathbf{y}})$$

Loss function example

```
def cross_entropy(y_true, y_pred, eps=1e-12):
    ...
    Log loss is undefined for p=0 or p=1,
        so probabilities are
        clipped to (eps, 1 - eps).
    ...
    y_true = np.clip(y_true, eps, 1. - eps)
    y_pred = np.clip(y_pred, eps, 1. - eps)
    log_likelihood = -np.log(y_pred)
    loss = -(y_true * np.log(y_pred)).sum()
    return loss

cross_entropy([1, 0, 0, 0], [0.9, 0, 0, 0.1])
0.10536051571528555

cross_entropy([1, 0, 0, 0], [0.1, 0.8, 0.1, 0])
2.3025850930218996
```

Loss function for a dataset of size N

$$J = -\frac{1}{N} \left(\sum_{i=1}^N \mathbf{y}_i \cdot \log(\hat{\mathbf{y}}_i) \right)$$

Evaluation metrics

total N=150	Predicted YES	Predicted NO
Actual: YES	True positive TP = 80	False Negative FN = 25
Actual: NO	False Positive FP = 10	True negative TN = 35

$$\text{precision} = \frac{\text{number of correct positive predictions}}{\text{number of positive predictions made}}$$

$$= \frac{TP}{TP + FP} = \frac{80}{80 + 10} = 0.89$$

$$\text{accuracy} = \frac{\text{number of correct predictions}}{\text{total number of predictions made}}$$

$$= \frac{TP + TN}{N} = \frac{80 + 35}{150} = 0.7$$

$$\text{recall} = \frac{\text{number of correct positive predictions}}{\text{number of all positive samples}}$$

$$= \frac{TP}{TP + FN} = \frac{80}{80 + 25} = 0.76$$

$$F1 = 2 * \frac{1}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}}$$

Harmonic mean between precision and recall

Evaluation metrics for multi-class classification

	Predicted YES	Predicted NO
Actual: YES	True positive	False Negative
Actual: NO	False Positive	True negative

$$\text{Precision} = \text{TP} / \text{Total predicted positive}$$

$$\text{Recall} = \text{TP} / \text{Total actual positive}$$

	GoldLabel_A	GoldLabel_B	GoldLabel_C	
Predicted_A	30	20	10	TotalPredicted_A=60
Predicted_B	50	60	10	TotalPredicted_B=120
Predicted_C	20	20	80	TotalPredicted_C=120
TotalGoldLabel_A=100	TotalGoldLabel_B=100	TotalGoldLabel_C=100		

$$\text{Precision label X} = \text{TP}_X / \text{Total predicted}_X$$

$$\text{Recall label X} = \text{TP}_X / \text{Total actual}_X$$

Ref: <http://text-analytics101.rxnlp.com/2014/10/computing-precision-and-recall-for.html>

sklearn.metrics

```
sklearn.metrics.precision_score(y_true, y_pred, labels=None, pos_label=1, average='binary', sample_weight=None)
```

`average : string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']`

This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

`'binary'`:

Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

`'micro'`:

Calculate metrics globally by counting the total true positives, false negatives and false positives.

`'macro'`:

Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

`'weighted'`:

Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

`'samples'`:

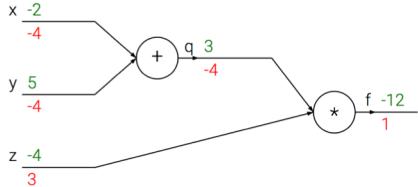
Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from `accuracy_score`).

Understanding the backpropagation

Backpropagation

- Forward -> compute loss -> loss run backward to find gradients -> update weights

$$f(x,y,z) = (x+y) * z$$



```

# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
# now backprop through q = x + y
dqdx = 1.0
dqdy = 1.0
dfdx = dfdq * dqdx # will be z = -4 #And the multiplication here is the chain rule!
dfdy = dfdq * dqdy # will be z = -4 #
print("(dfdx, dfdy, dfdz) = ({}, {}, {})".format(dfdx, dfdy, dfdz))
[ dfdx, dfdy, dfdz] = [-4.0, -4.0, 3]
  
```

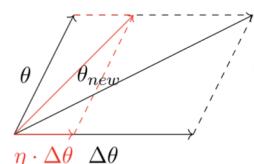
Credit: Kapathy (cs231n)

Updating parameters (1/2)

- Loss function w.r.t to $\theta = [w,b]$ where w and b are weights and biases

$$L = -y \cdot \log(\hat{y}) \quad \text{We want to find best } \theta \text{ that minimizes } L \rightarrow \text{How?}$$

- For a given θ , let's change it a small amount $\Delta\theta = u$
i.e., $\theta' = \theta + \Delta\theta$
Even better, using a controllable parameter (learning rate)
 $\theta' = \theta + \eta \cdot \Delta\theta$



- What will happen to L ?

Using Taylor series: $\nabla L(\theta) = [\partial L(\theta)/\partial w, \partial L(\theta)/\partial b]$

$$\mathcal{L}(\theta + \eta u) = \mathcal{L}(\theta) + \eta * u^T \boxed{\nabla_{\theta} \mathcal{L}(\theta)} + \frac{\eta^2}{2!} * u^T \nabla^2 \mathcal{L}(\theta) u + \frac{\eta^3}{3!} * \dots + \frac{\eta^4}{4!} * \dots$$

Updating parameters (2/2)

$$\mathcal{L}(\theta + \eta u) = \mathcal{L}(\theta) + \eta * u^T \nabla_{\theta} \mathcal{L}(\theta) + \underbrace{\frac{\eta^2}{2!} * u^T \nabla^2 \mathcal{L}(\theta) u + \frac{\eta^3}{3!} * \dots + \frac{\eta^4}{4!} * \dots}_{\eta \ll 1 \rightarrow \text{can ignore these values}}$$

When updating θ , we want the loss decreases
 → This means

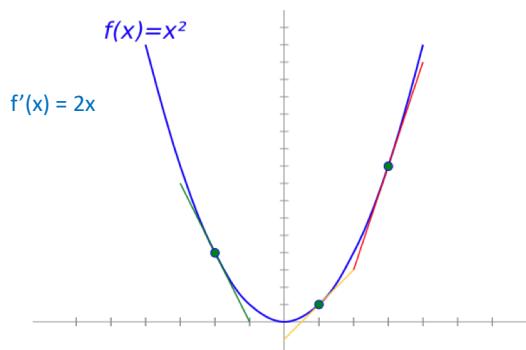
$$u^T \nabla_{\theta} \mathcal{L}(\theta) < 0$$

The angle between two vectors

$$-1 \leq \cos(\beta) = \frac{u^T \nabla_{\theta} \mathcal{L}(\theta)}{\|u\| * \|\nabla_{\theta} \mathcal{L}(\theta)\|} \leq 1$$

$u \cdot \nabla \mathcal{L}(\theta)$ is minimum when vector u and gradient is opposite
 i.e., $\cos(\beta) = -1$ or $\beta = 180^\circ$ with respect to the gradient

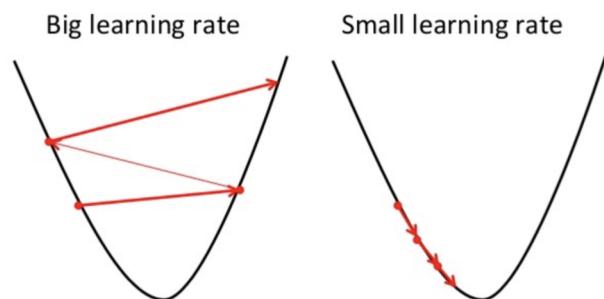
An intuitive example of gradient descent



$x < 0 \rightarrow f' < 0 \rightarrow$ the function is decreasing
 Update x so that in the opposite of current x
 → $x - e$, e has same sign with f' (< 0)
 → x goes to 0

$x > 0 \rightarrow f' > 0 \rightarrow$ the function is increasing
 Update x so that in the opposite of current x
 → $x - e$, e has same sign with f' (> 0)
 → x goes to 0

Learning rate



Gradient descent

The simplest updating rule using gradient

$$w_{t+1} = w_t - \eta \nabla w_t$$

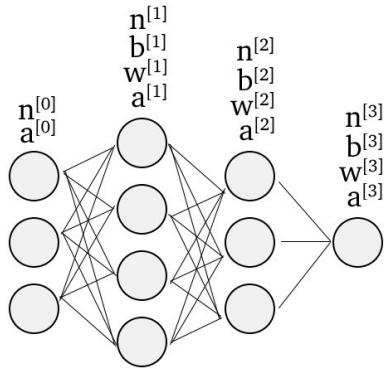
$$b_{t+1} = b_t - \eta \nabla b_t$$

$$\text{where, } \nabla w_t = \frac{\partial \mathcal{L}(w, b)}{\partial w} \text{ at } w = w_t, b = b_t, \quad \nabla b = \frac{\partial \mathcal{L}(w, b)}{\partial b} \text{ at } w = w_t, b = b_t$$

In order to update the parameters, we need to compute all the partial derivatives with respect to the components of θ (i.e., weights w and biases b)

HOW?

Calculating derivation



$n^{[l]}$ - The number of neurons in the layer l .
 $w^{[l]}$ - The weight matrix associated with the layers l and $l-1$, of the size $(n^{[l]} \times n^{[l-1]})$.

$w_{jk}^{[l]}$ refers to the weight associated with the neuron j in the layer l and the neuron k in the layer $l-1$.

$b^{[l]}$ - The vector of biases associated with the layer l , of the size $(n^{[l]} \times 1)$.

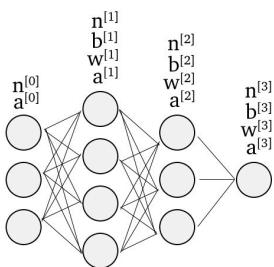
$a^{[l]}$ - The vector of activations of the neurons in the layer l , of the size $(n^{[l]} \times 1)$.

$z^{[l]}$ - The vector of the weighted output of the neurons in the layer l , of the size $(n^{[l]} \times 1)$.

$g^{[l]}$ - The activation function applied to the output of the neurons in the layer l , $a^{[l]} = g^{[l]}(z^{[l]})$.

Source: <https://medium.com/binaryandmore/beginners-guide-to-deriving-and-implementing-backpropagation-e3c1a5a1e536>

Calculating derivation



Forward propagation

$$z^{[l]} = w^{[l]} \cdot a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]})$$

Loss (Cost) function

$$C = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

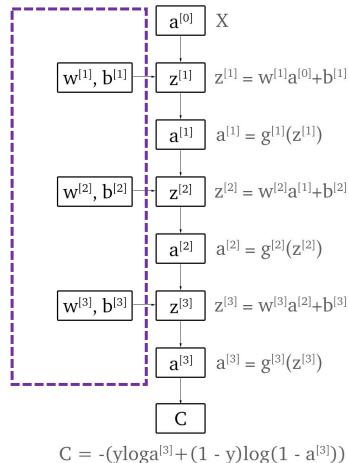
Update rules

$$w^{[l]} = w^{[l]} - \alpha \frac{\partial C}{\partial w^{[l]}}$$

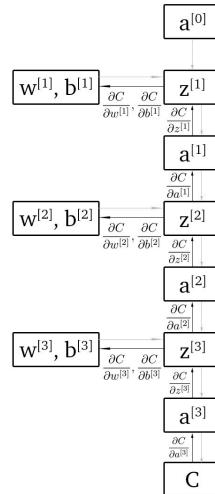
$$b^{[l]} = b^{[l]} - \alpha \frac{\partial C}{\partial b^{[l]}}$$

Computation graph

Need to compute the partial derivative of C wrt learnable parameters



Chain rules



$$\frac{\partial C}{\partial w^{[1]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial w^{[1]}}$$

$$\frac{\partial C}{\partial b^{[1]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial b^{[1]}}$$

$$\frac{\partial C}{\partial w^{[2]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}}$$

$$\frac{\partial C}{\partial b^{[2]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial b^{[2]}}$$

$$\frac{\partial C}{\partial w^{[3]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}}$$

$$\frac{\partial C}{\partial b^{[3]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial b^{[3]}}$$

Automatic differentiation: Autograd

Autograd is a PyTorch package for the differentiation for all operations on Tensors.
It performs the backpropagation starting from a variable (i.e., the tensor of the loss function).

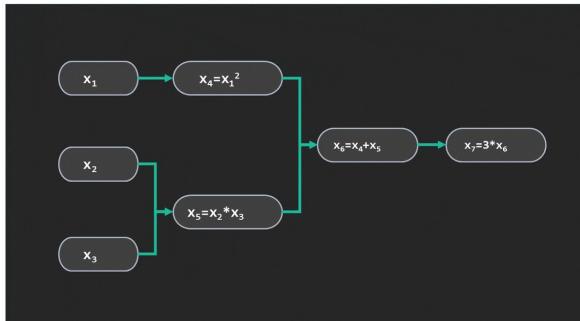
`loss.backward()`

This executes the backward pass and computes all the backpropagation gradients automatically.

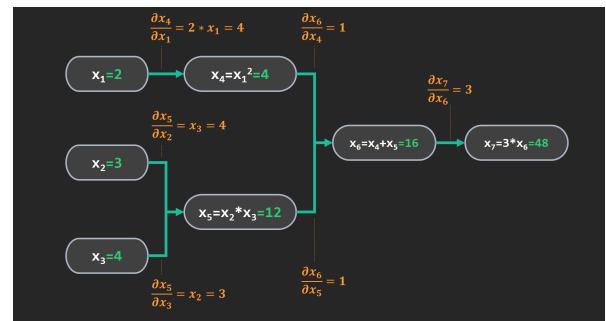
`w.grad`

We access individual gradient through the attributes `grad` of a variable.

How autodiff works?

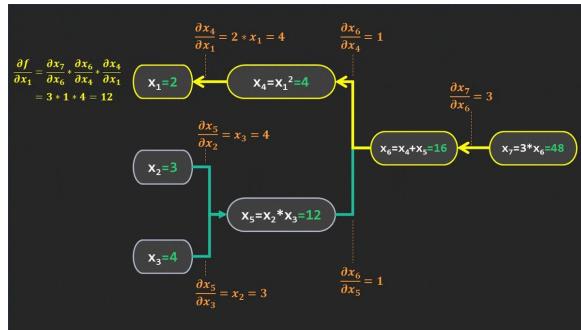


Forward pass



Partial derivatives of each node wrt its inputs

How autodiff works?

Ref: <https://medium.com/@rhome/automatic-differentiation-26d5a993692b>

Optimizers

What is an optimizer?

Prediction

$$\hat{y} = f(x; w, b) = f(x; \theta) \text{ where } \theta = [w, b]$$

w: weights
b: biases

Loss function

$$L(\theta) = -y \cdot \log(\hat{y})$$

Optimizers are algorithms/methods that change the attributes of the model in order to reduce the objective function

Objective:

$$\min_{\theta} L(\theta)$$

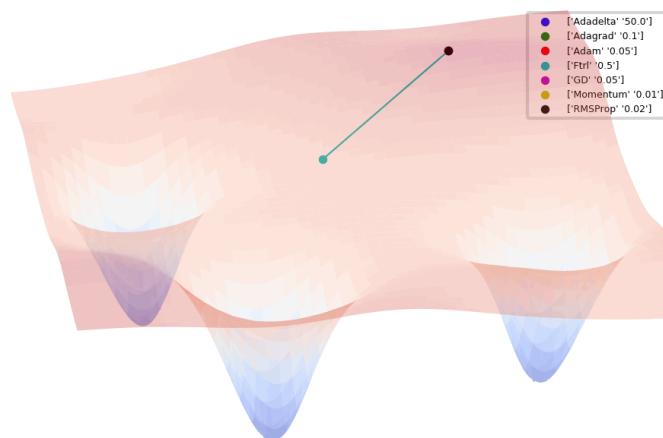


How?

Start from a random position of theta
Iteratively update it

$$\theta = \theta + \eta \Delta \theta$$

Iterative weight updating



Visualization credit: <https://github.com/Jaewan-Yun/optimizer-visualization>

Optimizers

- Gradient descent (GD)
- Stochastic gradient descent (SGD)
- Mini-batch gradient descent
- Momentum gradient descent
- Adaptive gradient descent (AdaGrad)
- Root mean square propagation gradient descent (RMSProp)
- Adaptive moment estimation (Adam)

Gradient descent update rule

- Randomly initialize w,b
- Iterate over data
 - Compute y-hat
 - Compute loss L(w,b)
 - Update

$$w_{t+1} = w_t - \eta \nabla w_t$$

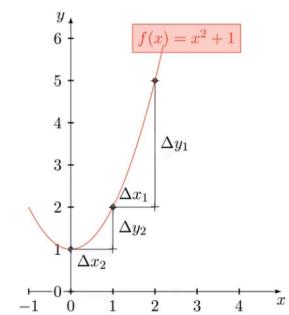
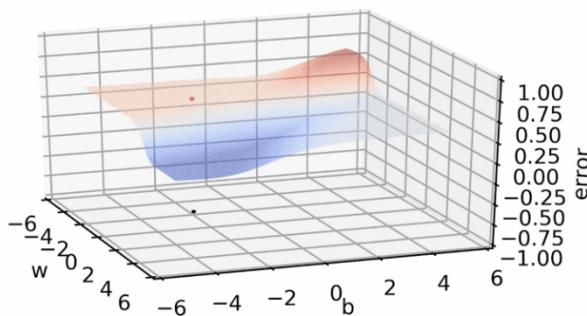
$$b_{t+1} = b_t - \eta \nabla b_t$$

where, $\nabla w_t = \frac{\partial \mathcal{L}(w, b)}{\partial w}$ at $w = w_t, b = b_t$, $\nabla b = \frac{\partial \mathcal{L}(w, b)}{\partial b}$ at $w = w_t, b = b_t$

Number of updates in one epoch

- Batch gradient descent → 1
- Stochastic gradient descent → N (N = number of data samples)
- Mini-batch gradient descent → N/B (B = batch size)

Gradient descent in flat vs steep area



Visualization credit: Niranjan Kumar

Momentum

Go faster if the same direction is repeated

$$v_t = \boxed{\gamma * v_{t-1}} + \eta \nabla w_t$$

$$w_{t+1} = w_t - v_t$$

AdaGrad (adaptive learning rate)

- Perform larger updates for infrequent parameters and smaller updates for frequent parameters
 - Parameter that is not zero most of the times -> small learning rate
 - Parameter that is zero most of the times, when it is on → higher learning rate to boost the gradient update

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

Non-sparse parameters will have large history value due to frequent updates
 → Small learning rate

Root Mean Square Propagation (RMSProp)

Decaying the history to prevent the rapid growth of the denominator in AdaGrad

$$v_t = \beta * v_{t-1} + (1 - \beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

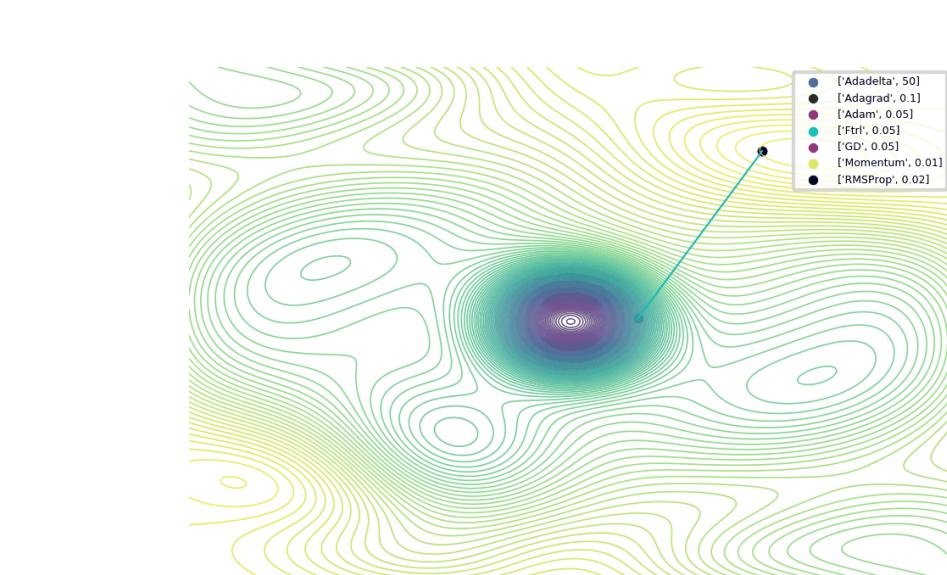
Adaptive moment estimation (Adam)

- Combine Momentum & RMSProp

$$m_t = \beta_1 * v_{t-1} + (1 - \beta_1)(\nabla w_t) \quad \rightarrow \text{use history to compute update}$$

$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2)(\nabla w_t)^2 \quad \rightarrow \text{use history to adjust learning rate (shrink or boost)}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} m_t$$



Visualization credit: <https://github.com/Jaewan-Yun/optimizer-visualization>

Learning hyperparameters

- Learning rate
- Momentum
- Decay
- Epsilon (fuzz factor)
- Beta_1, beta_2 -> Adam

References

- Intuition of gradient descent
 - <https://hackernoon.com/the-reason-behind-moving-in-the-direction-opposite-to-the-gradient-f9566b95370b>
- Backpropagation
 - <https://medium.com/binaryandmore/beginners-guide-to-deriving-and-implementing-backpropagation-e3c1a5a1e536>
- Optimizers:
 - <https://hackernoon.com/demystifying-different-variants-of-gradient-descent-optimization-algorithm-19ae9ba2e9bc>