[Live] 실전! TensorFlow로 배우는 딥러닝

Day2

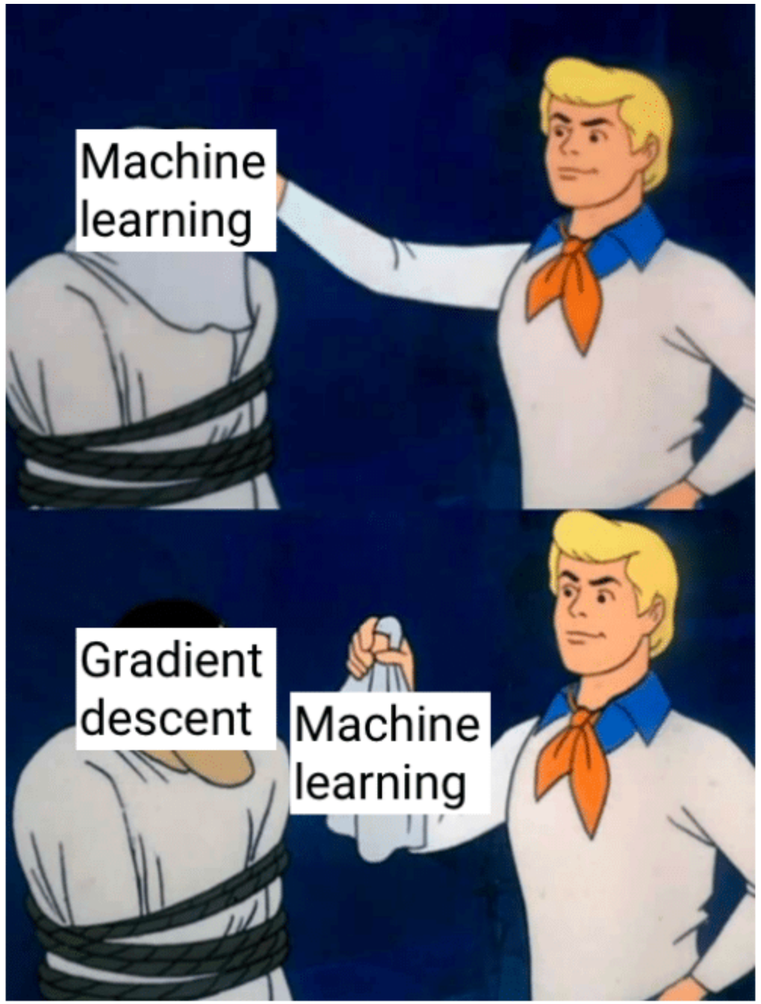
- Optimization & Regularization

- CNN

목차

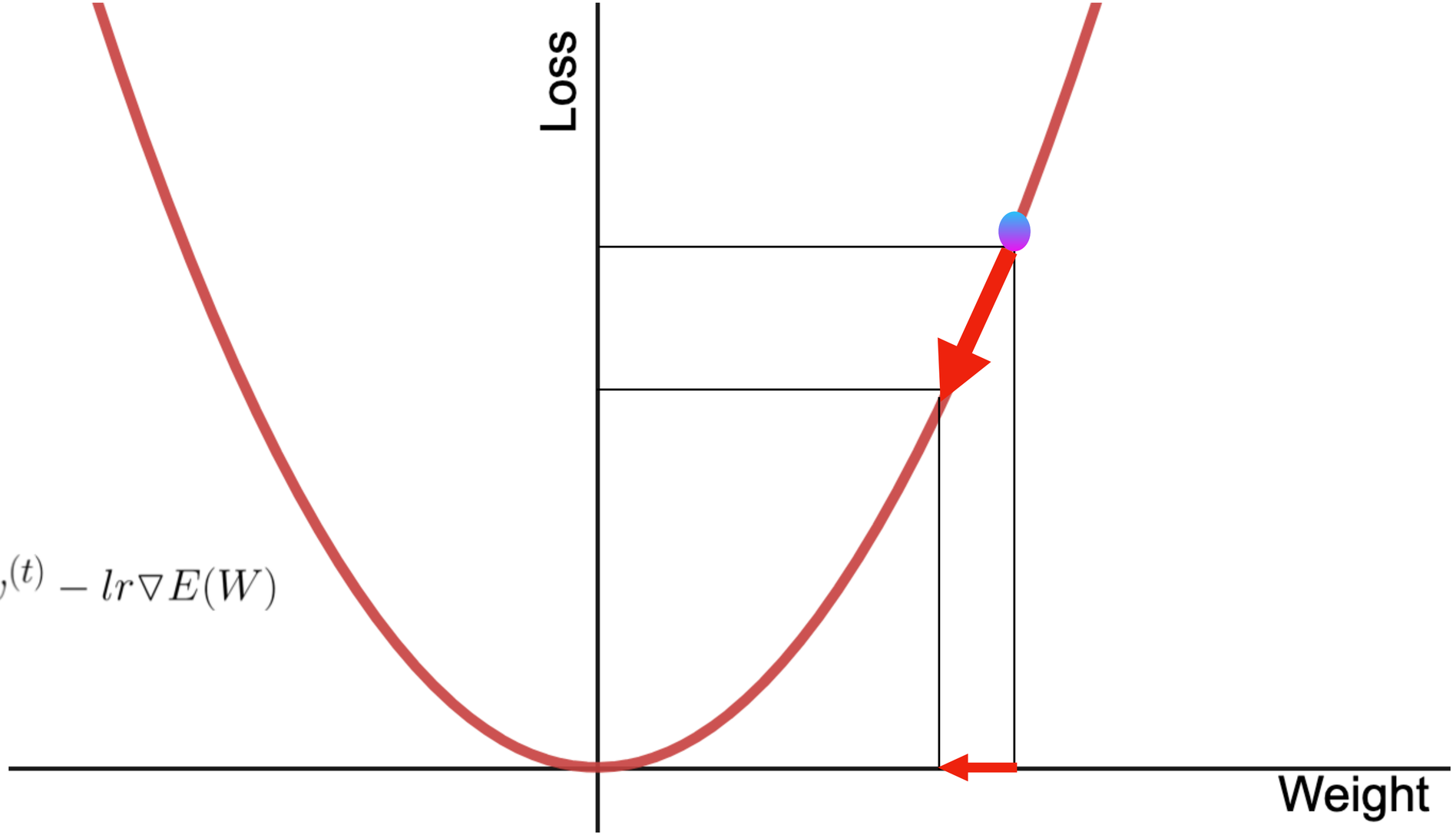
1. Optimization
   - ✓ Optimization
   - ✓ Gradient Descent
   - ✓ SGD + Momentum
   - ✓ 다양한 최적화 함수들
2. Generalization
   - ✓ Initialization
   - ✓ Over-fitting
   - ✓ Early stopping
   - ✓ Regularization
   - ✓ Dropout
   - ✓ Data Augmentation
   - ✓ Batch Normalization
3. Tensorflow를 이용한 DNN (Fashion MNIST Classification)
4. CNN
5. Tensorflow를 이용한 CNN (Fashion MNIST Classification)

1. Optimization
optimization



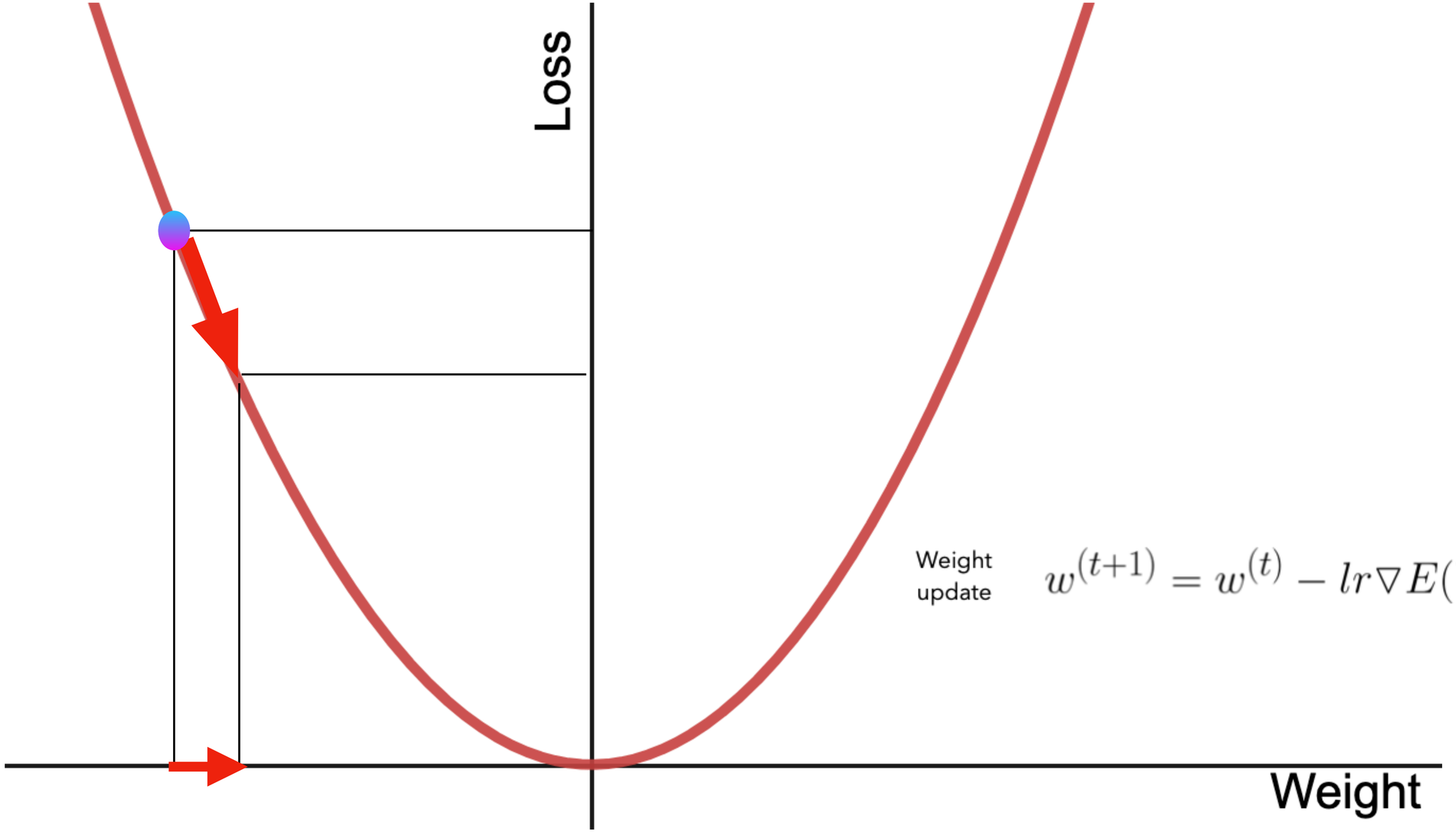
Machine learning behind the scenes

1. Optimization
optimization



Weight update $w^{(t+1)} = w^{(t)} - lr \nabla E(W)$

1. Optimization
optimization

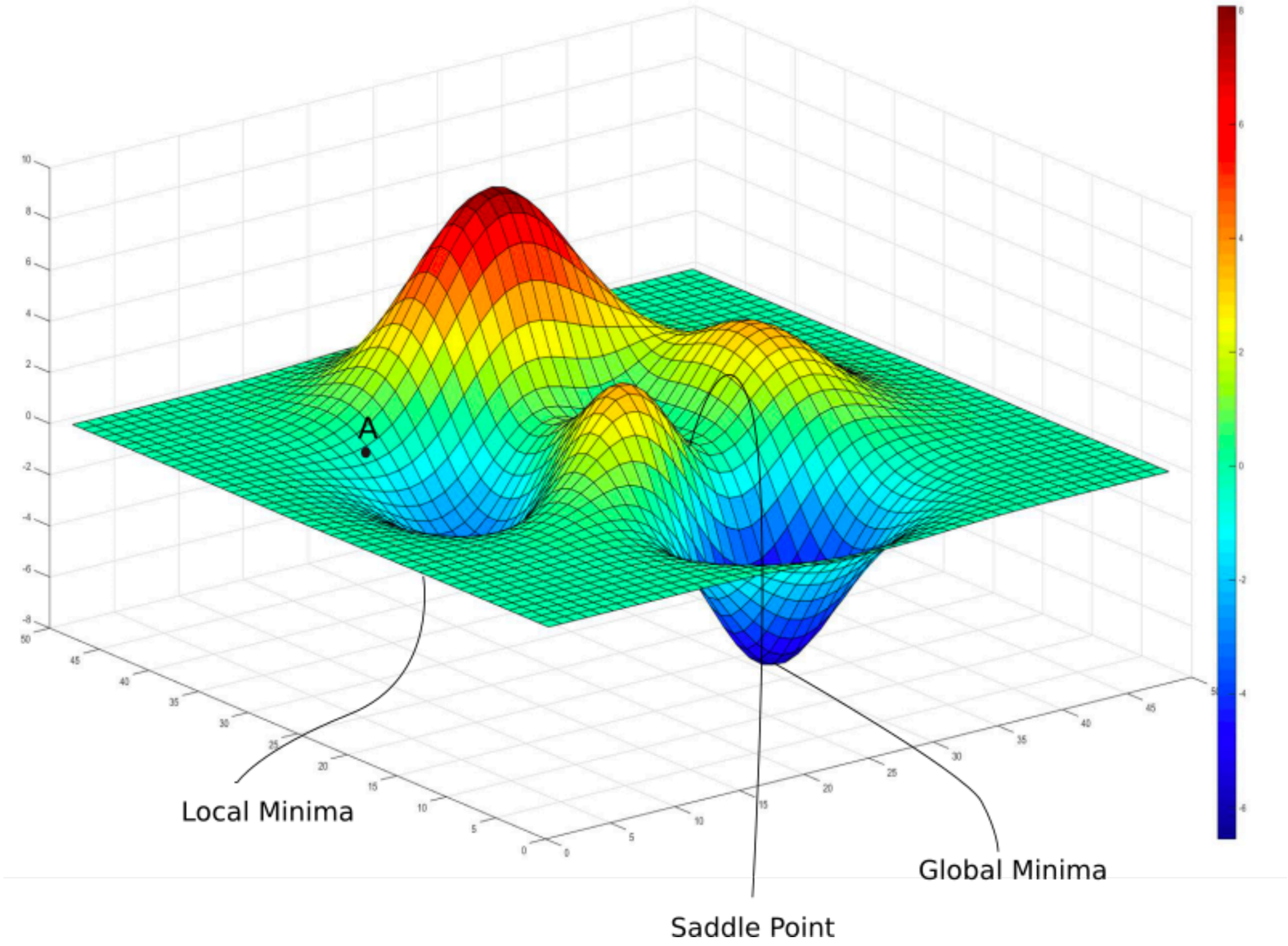


Weight update $w^{(t+1)} = w^{(t)} - lr \nabla E(W)$

그런데,
정말 Loss function은 weight space에서
이쁜 빗살무늬토기 모양일까??

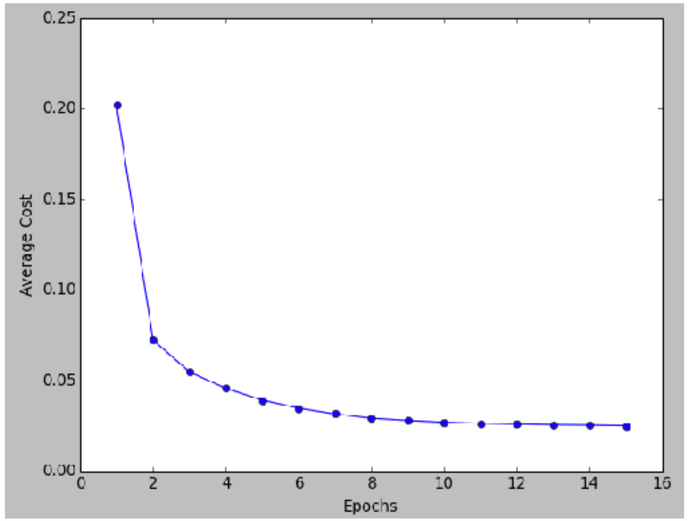
https://memory.library.kr/items/show/31795

1. Optimization
optimization



gradient descent local minima

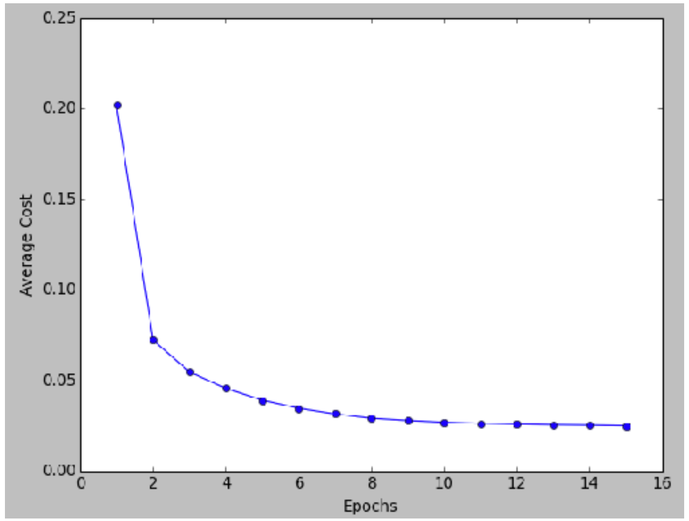
Batch Gradient descent



$$E(W) = \frac{1}{2}\sum_{n=1}^{N}(t - \hat{y}_n)^2$$

1. 모든 data에 대해서 loss를 다 구한다.
2. 개별 input에 대하여 구한 loss를 다 더하여 데이터 개수로 나눠준다.(평균을 구한다.)
3. Weight에 대한 gradient를 구한다.
4. backpropagation으로 한 번 weight가 update된다.

Batch Gradient descent



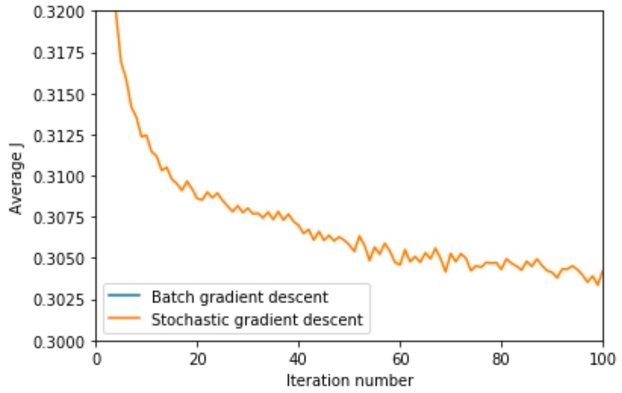
$$E(W) = \frac{1}{2}\sum_{n=1}^{N}(t - \hat{y}_n)^2$$

1.
2. 개별 input 구한다.)
3.
4. backpropagation으로 한 번 weight가 update된다.

만약, 데이터가 50만개라면?

50만번의 계산을 해야 가중치가 한 번 update된다.

Stochastic Gradient descent

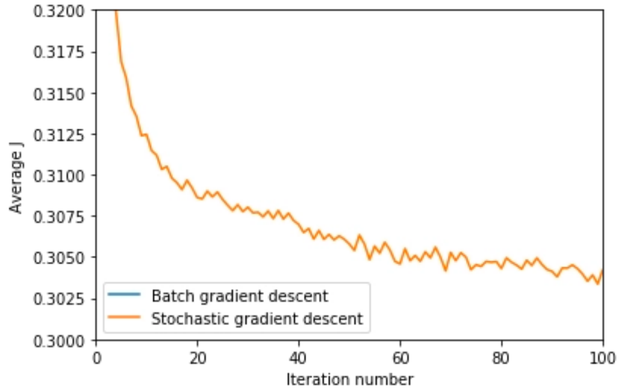


$$E_n(W) = \frac{1}{2}(t - \hat{y}_n)^2$$
$$n = 1 \; to \; N$$

1. Single sample로 loss를 계산한다.
2. Weight에 대한 gradient를 구한다.
3. backpropagation으로 한 번 weight가 update된다.
4. Updated weight로 다음 sample data에 대한 loss를 구한다.
5. 2~4번을 전체 데이터에 대하여 반복한다.

https://adventuresinmachinelearning.com/stochastic-gradient-descent/

Stochastic Gradient descent



$$E_n(W) = \frac{1}{2}(t - \hat{y}_n)^2$$
$$n = 1 \; to \; N$$

속도는 빠르지만,

data의 noise 때문에 학습이 불안정하고

BGD 보다 성능이 떨어진다.

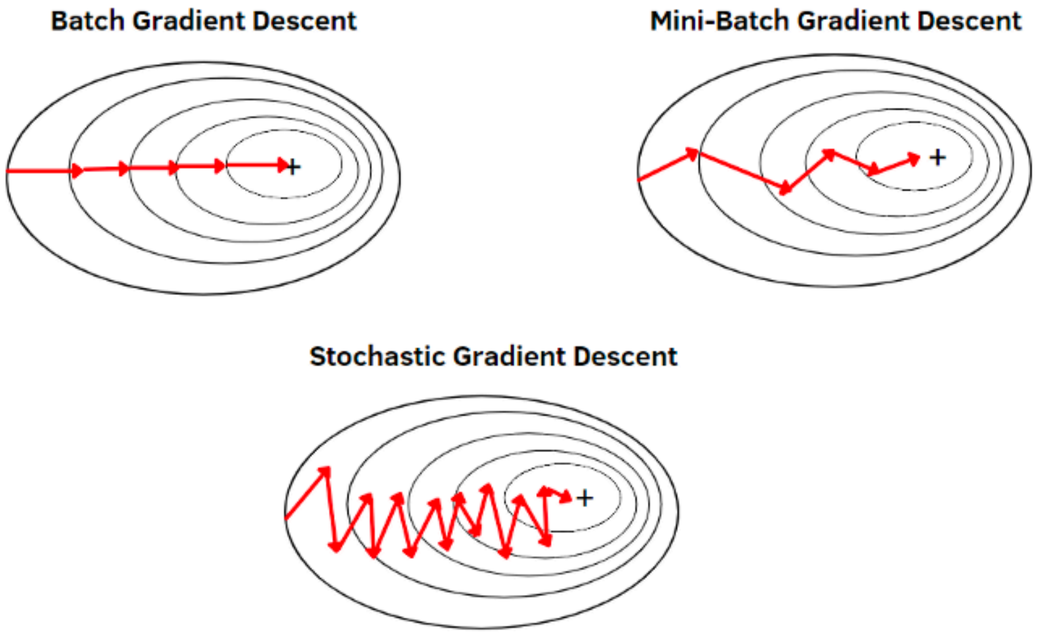
1.
2.
3.
4. Updated weight로 다음 sample data에 대한 loss를 구한다.
5. 2~4번을 전체 데이터에 대하여 반복한다.

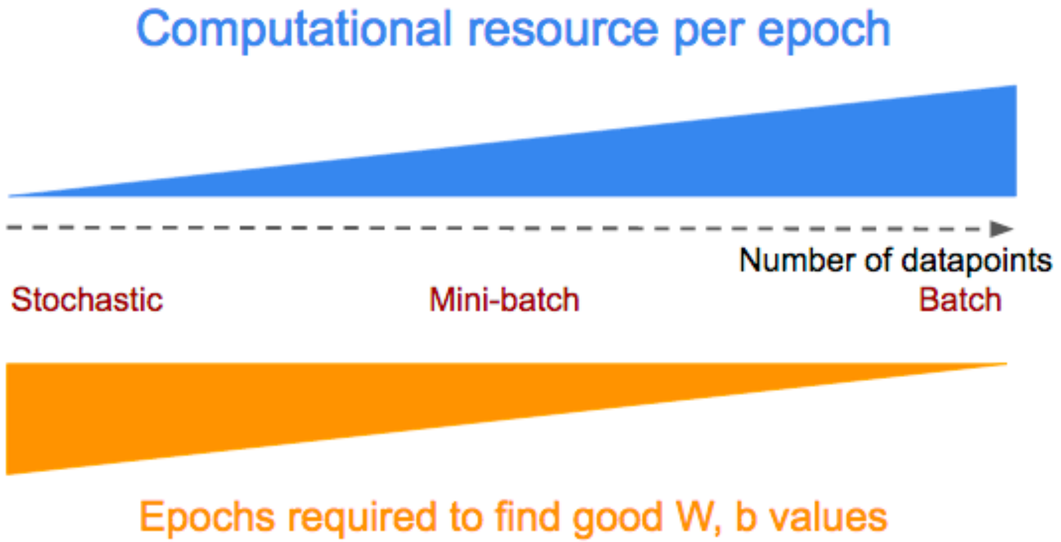
Mini-Batch Gradient descent

$$E_k(W) = \frac{1}{2}\sum_{m=1}^{M}(t - \hat{y}_m)^2$$
$$k = 1\ to\ K\ \ (\texttt{number of mini-batch})$$
$$M = N/K\ \ (\texttt{size of single mini-batch})$$

1. Mini batch size 만큼 loss를 계산한다.
2. Weight에 대한 gradient를 구한다.
3. backpropagation으로 한 번 weight가 update된다.
4. Updated weight로 다음 mini batch samples에 대한 loss를 구한다.
5. 2~4번을 전체 데이터에 대하여 반복한다.

Put all GDs together

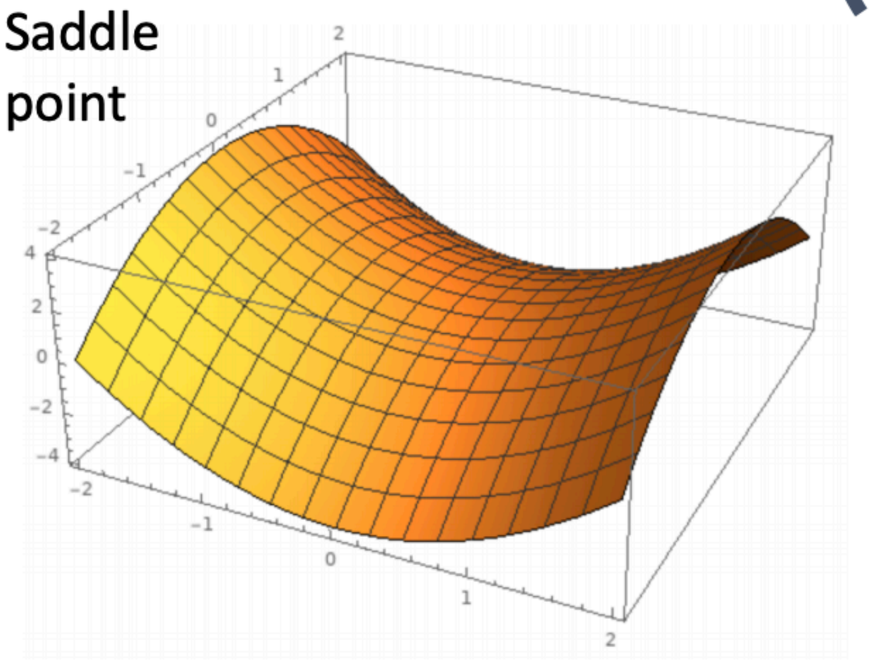




https://www.kdnuggets.com/2016/08/gentlest-introduction-tensorflow-part-2.html/2

Problems with SGD

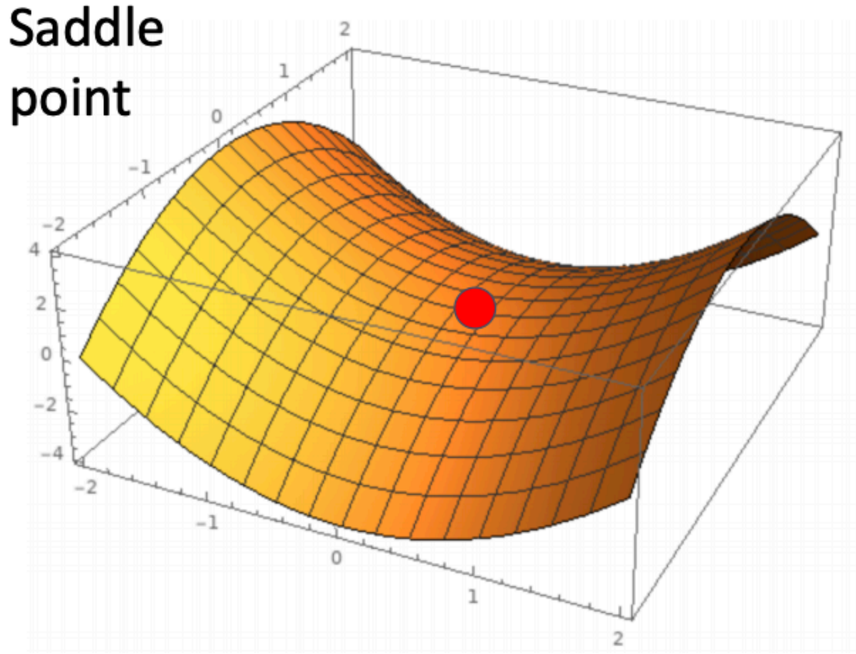
What if the loss function has a **local minimum** or **saddle point**?



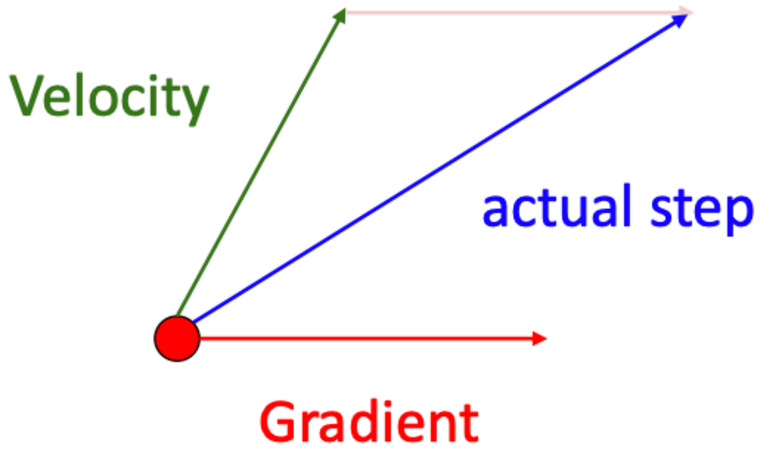
Problems with SGD

What if the loss function has a **local minimum** or **saddle point**?

Zero gradient, gradient descent gets stuck



Momentum update:



Combine gradient at current point with velocity to get step used to update weights

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

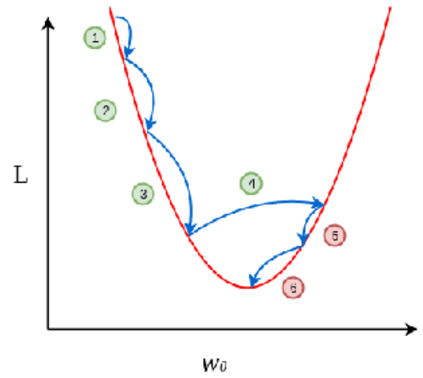
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

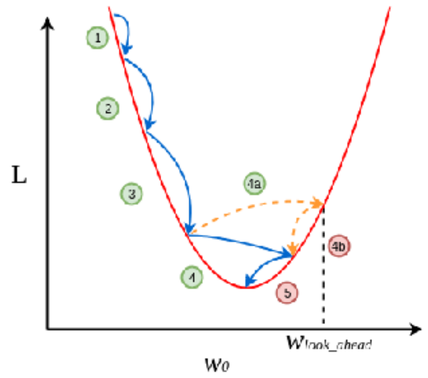
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$



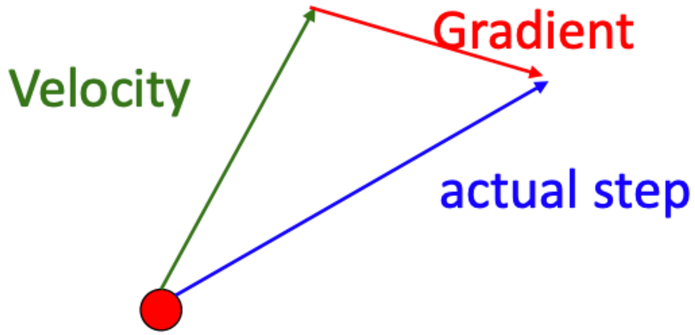
(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

$\Rightarrow \frac{\partial L}{\partial w_0} - \frac{Negative(-)}{Positive(+)}$

$\Rightarrow \frac{\partial L}{\partial w_0} - \frac{Negative(-)}{Negative(-)}$



"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Learning Parameters, Part 2: Momentum-Based & Nesterov Accelerated Gradient Descent

Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  old_v = v
  v = rho * v - learning_rate * dw
  w -= rho * old_v - (1 + rho) * v
```

AdaGrad

$$h \leftarrow h + \frac{\partial L}{\partial W} \odot \frac{\partial L}{\partial W}$$
$$W \leftarrow W - lr \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W}$$

AdaGrad

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\varepsilon I + diag(G_t)}} \cdot g_t, \tag{1}$$

$$g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t), \tag{2}$$

$$G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^\top. \tag{3}$$

$$\begin{bmatrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \vdots \\ \theta_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} \theta_t^{(1)} \\ \theta_t^{(2)} \\ \vdots \\ \theta_t^{(m)} \end{bmatrix} - \begin{bmatrix} \frac{\eta}{\sqrt{\varepsilon + G_t^{(1,1)}}} & 0 & \cdots & 0 \\ 0 & \frac{\eta}{\sqrt{\varepsilon + G_t^{(2,2)}}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\eta}{\sqrt{\varepsilon + G_t^{(m,m)}}} \end{bmatrix} \cdot \begin{bmatrix} g_t^{(1)} \\ g_t^{(2)} \\ \vdots \\ g_t^{(m)} \end{bmatrix}. \tag{5}$$

https://medium.com/konvergen/an-introduction-to-adagrad-f130ae871827

AdaGrad

$$\begin{bmatrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \vdots \\ \theta_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} \theta_{t}^{(1)} \\ \theta_{t}^{(2)} \\ \vdots \\ \theta_{t}^{(m)} \end{bmatrix} - \begin{bmatrix} \frac{\eta}{\sqrt{\varepsilon + G_t^{(1,1)}}} g_t^{(1)} \\ \frac{\eta}{\sqrt{\varepsilon + G_t^{(2,2)}}} g_t^{(2)} \\ \vdots \\ \frac{\eta}{\sqrt{\varepsilon + G_t^{(m,m)}}} g_t^{(m)}. \end{bmatrix} \tag{6}$$

$$\begin{bmatrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \vdots \\ \theta_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} \theta_{t}^{(1)} \\ \theta_{t}^{(2)} \\ \vdots \\ \theta_{t}^{(m)} \end{bmatrix} - \begin{bmatrix} \eta g_t^{(1)} \\ \eta g_t^{(2)} \\ \vdots \\ \eta g_t^{(m)}. \end{bmatrix}, \tag{7}$$

 https://medium.com/konvergen/an-introduction-to-adagrad-f130ae871827

RMSProp: "Leaky Adagrad"

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

Tieleman and Hinton, 2012

Adam

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Adam (almost) : RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

SGD+Momentum

Adam (almost) : RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

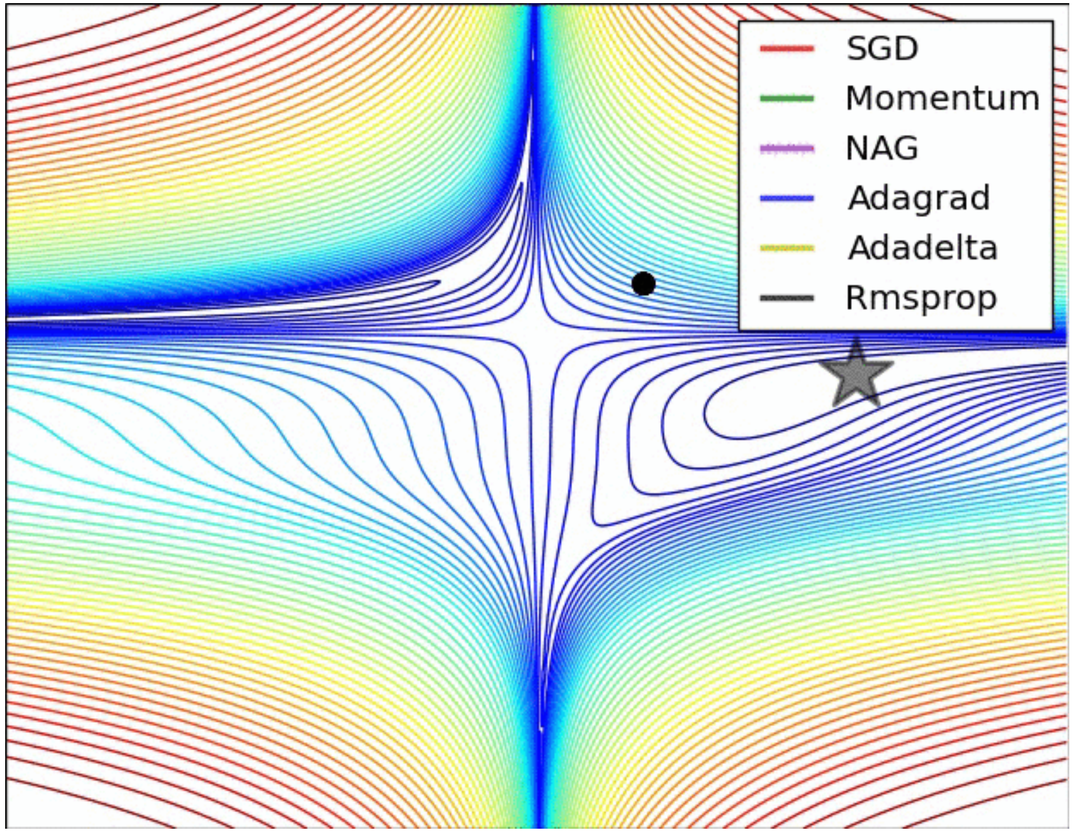
Momentum

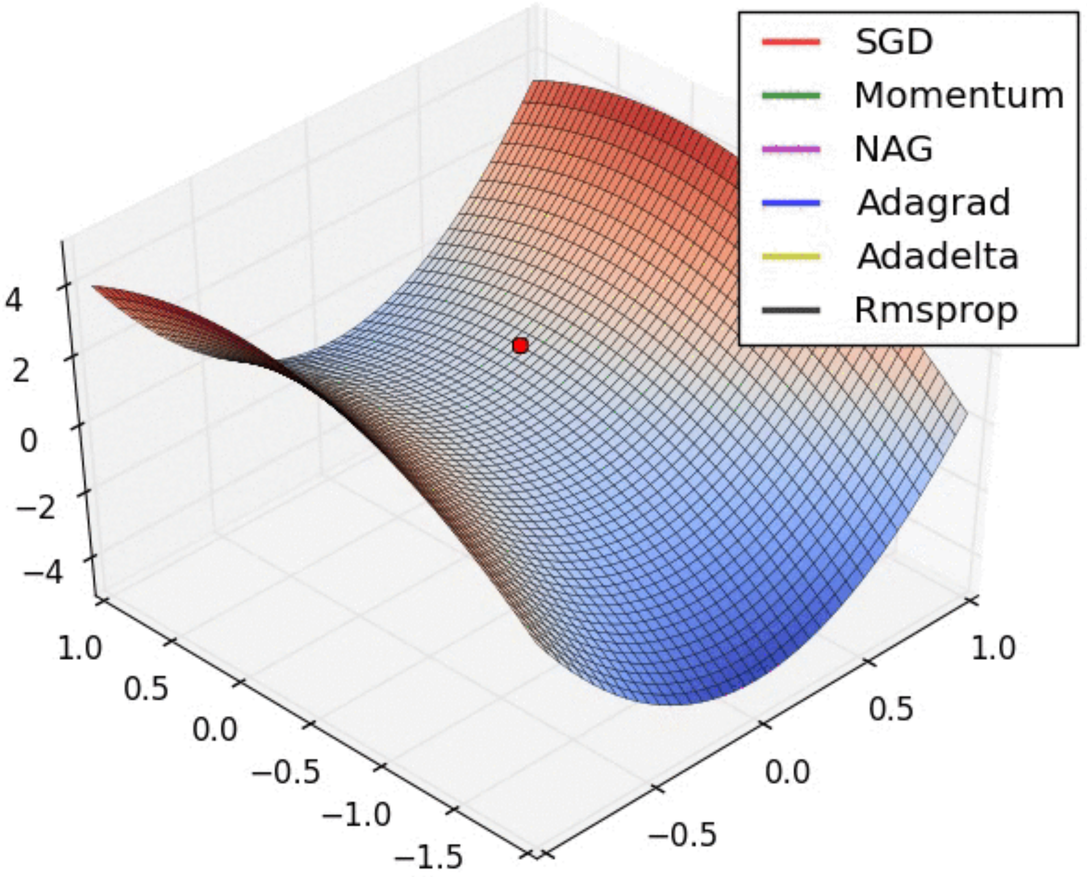
AdaGrad / RMSProp

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

1. Optimization
Better optimizer





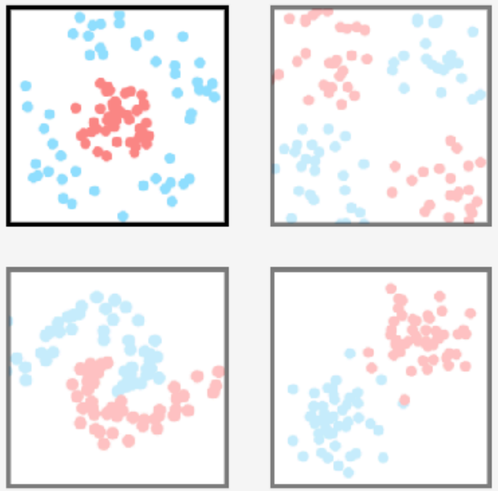
Optimization Algorithm Comparison

Algorithm	Tracks first moments (Momentum)	Tracks second moments (Adaptive learning rates)	Leaky second moments	Bias correction for moment estimates
SGD	✗	✗	✗	✗
SGD+Momentum	✓	✗	✗	✗
Nesterov	✓	✗	✗	✗
AdaGrad	✗	✓	✗	✗
RMSProp	✗	✓	✓	✗
Adam	✓	✓	✓	✓

2. Generalization initialization

1. Choose input dataset

Select a training dataset.



This legend details the color scheme for labels, and the values of the weights/gradients.

Label/Prediction: 0 0.5 1

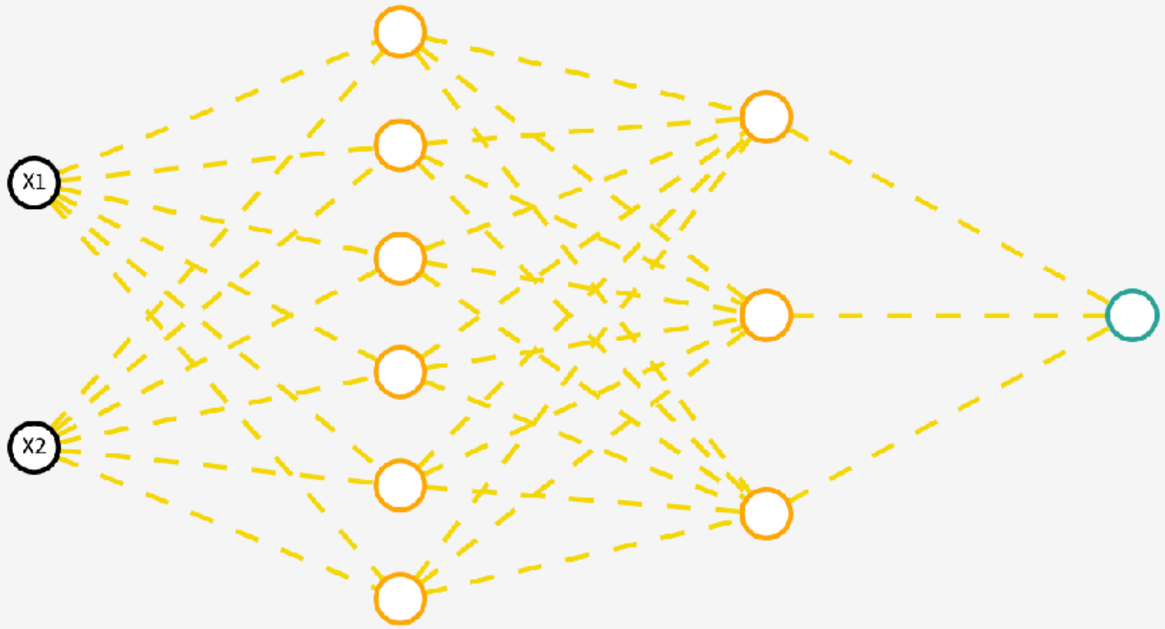
Weight/Gradient: neg zero pos

Node Type: Input Relu Sigmoid

2. Choose initialization method

Select an initialization method for the values of your neural network parameters[1].

Zero Too small Appropriate Too large

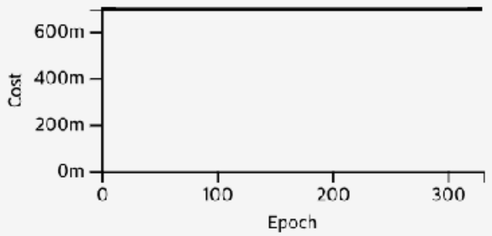


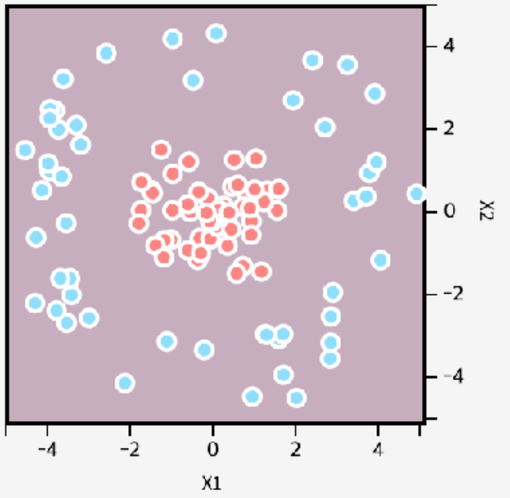
Select whether to visualize the weights or gradients of the network above.

Weight Gradient

3. Train the network.

Observe the cost function and the decision boundary.

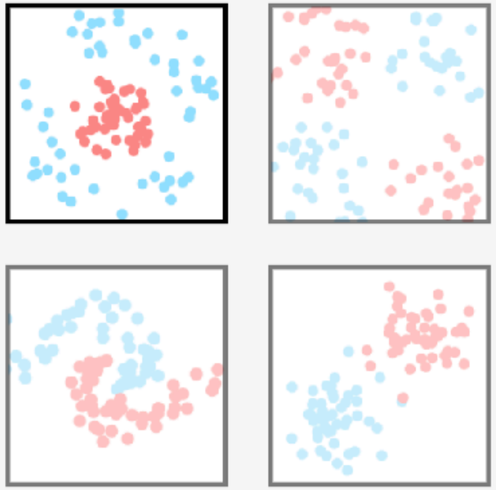




2. Generalization initialization

1. Choose input dataset

Select a training dataset.



This legend details the color scheme for labels, and the values of the weights/gradients.

Label/Prediction: 0 0.5 1

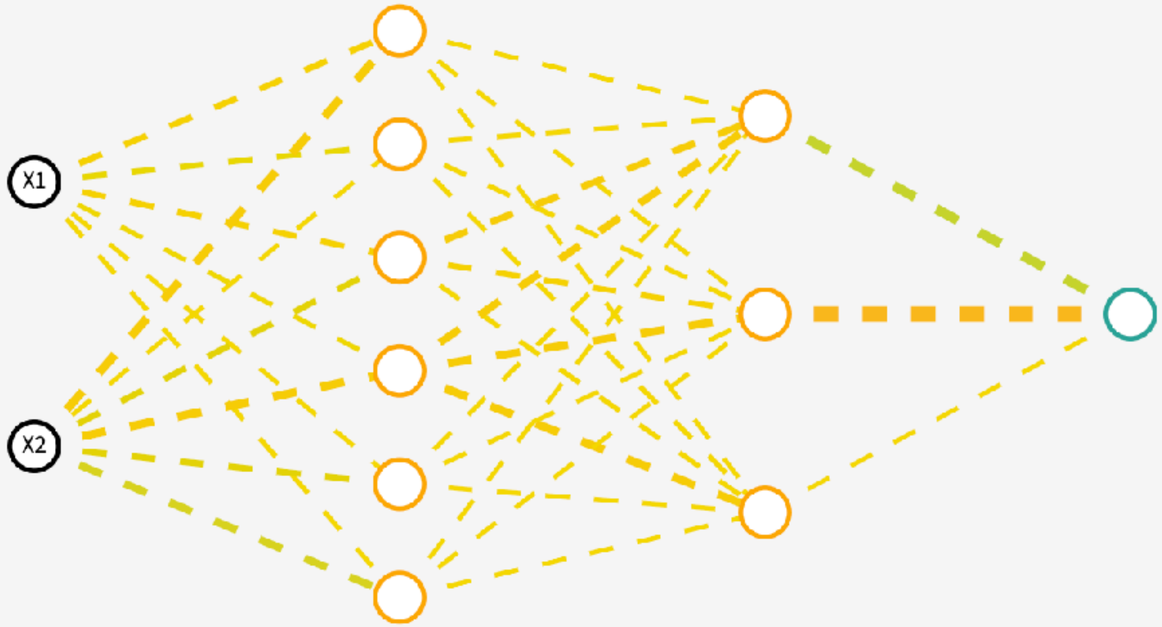
Weight/Gradient: neg zero pos

Node Type: Input Relu Sigmoid

2. Choose initialization method

Select an initialization method for the values of your neural network parameters[1].

Zero Too small Appropriate Too large

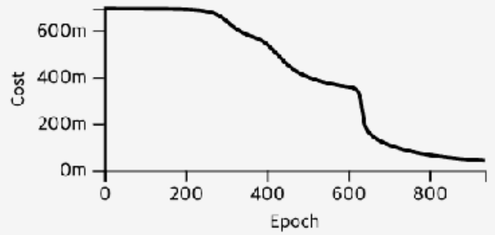


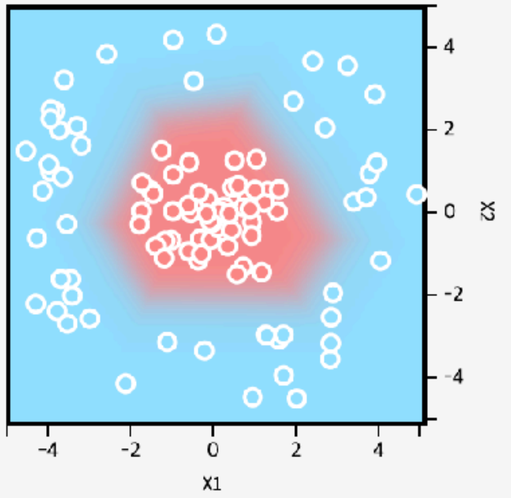
Select whether to visualize the weights or gradients of the network above.

Weight Gradient

3. Train the network.

Observe the cost function and the decision boundary.

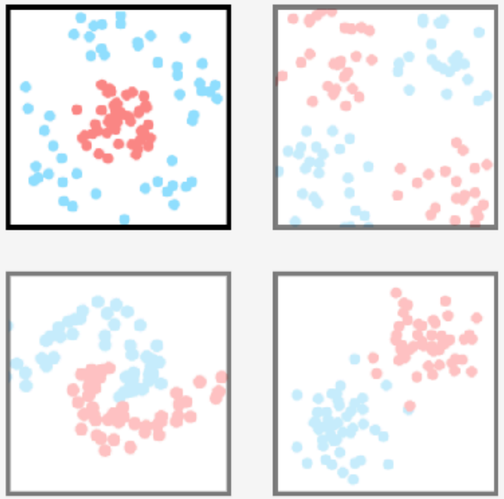




2. Generalization initialization

1. Choose input dataset

Select a training dataset.



This legend details the color scheme for labels, and the values of the weights/gradients.

Label/Prediction: 0 0.5 1

Weight/Gradient: neg zero pos

Node Type: Input Relu Sigmoid

2. Choose initialization method

Select an initialization method for the values of your neural network parameters[1].

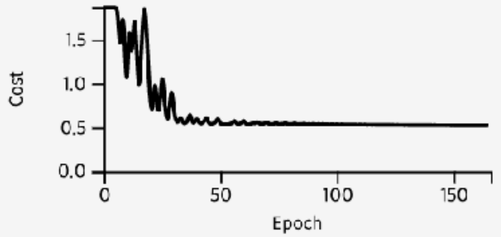
Zero Too small Appropriate Too large

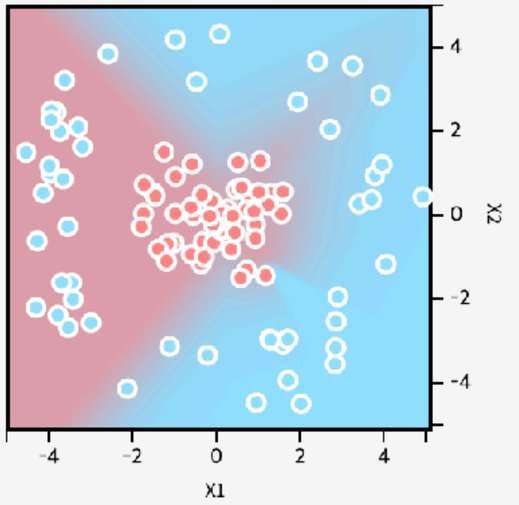
Select whether to visualize the weights or gradients of the network above.

Weight Gradient

3. Train the network.

Observe the cost function and the decision boundary.

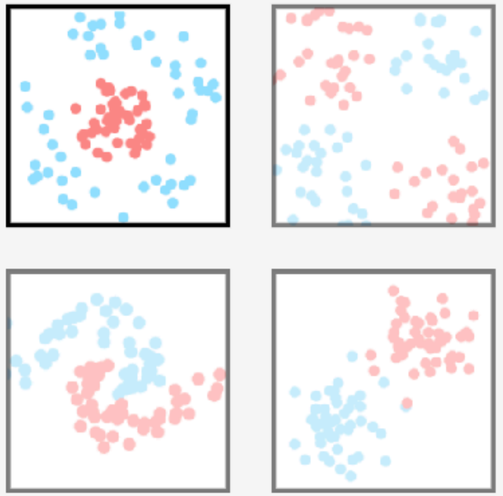




2. Generalization initialization

1. Choose input dataset

Select a training dataset.



This legend details the color scheme for labels, and the values of the weights/gradients.

Label/Prediction: 0 0.5 1

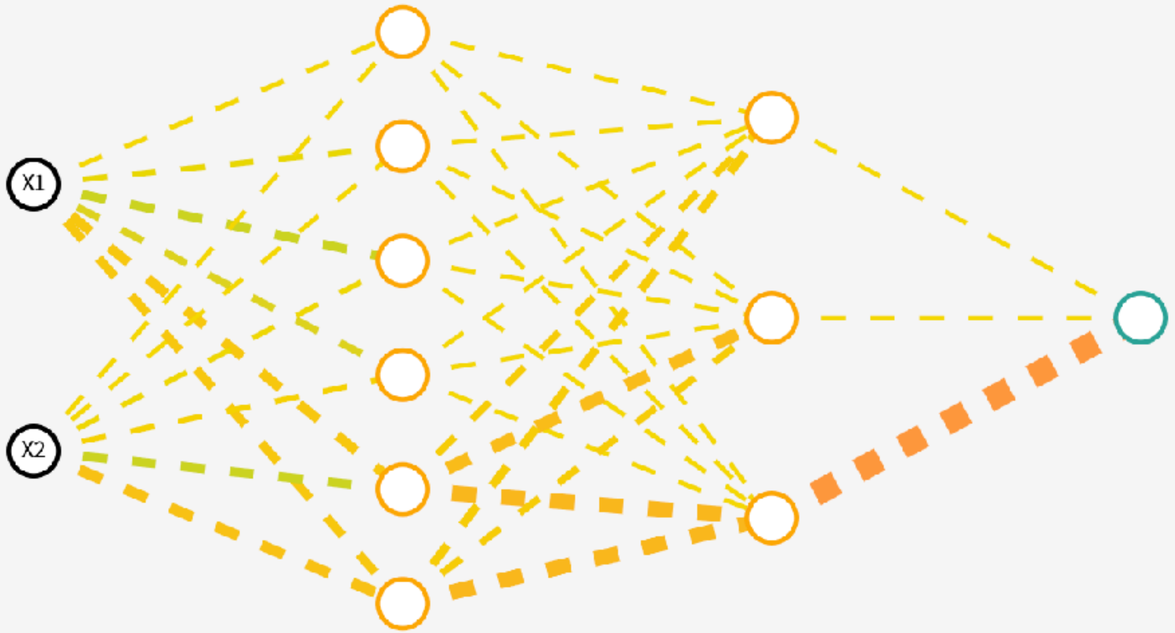
Weight/Gradient: neg zero pos

Node Type: Input Relu Sigmoid

2. Choose initialization method

Select an initialization method for the values of your neural network parameters[1].

Zero Too small Appropriate Too large

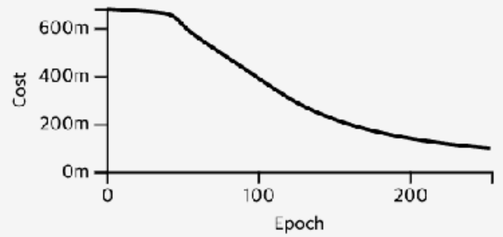


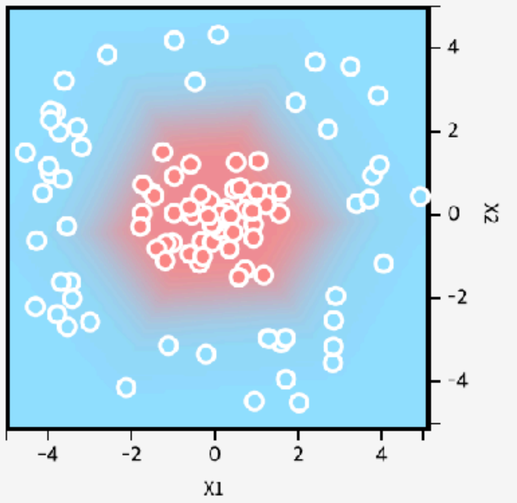
Select whether to visualize the weights or gradients of the network above.

Weight Gradient

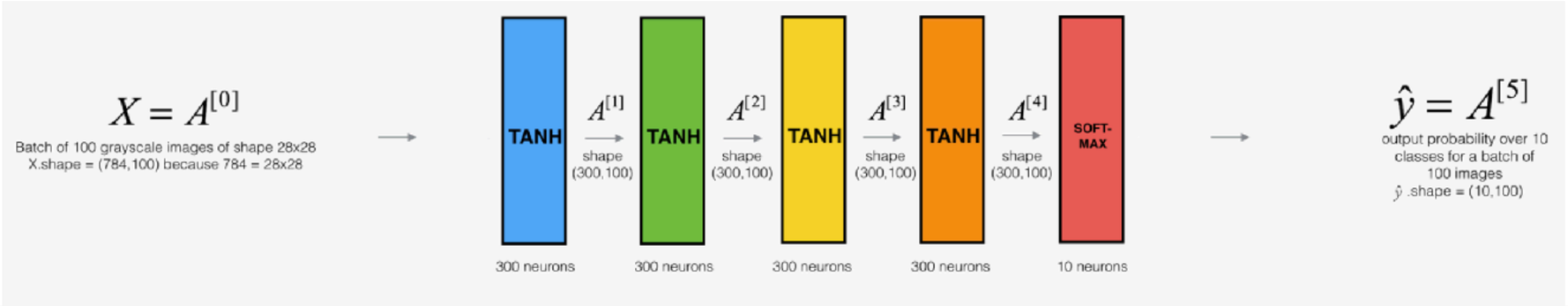
3. Train the network.

Observe the cost function and the decision boundary.





2. Generalization initialization



1. Load your dataset

Load 10,000 handwritten digits images (MNIST).

Load MNIST (100%)

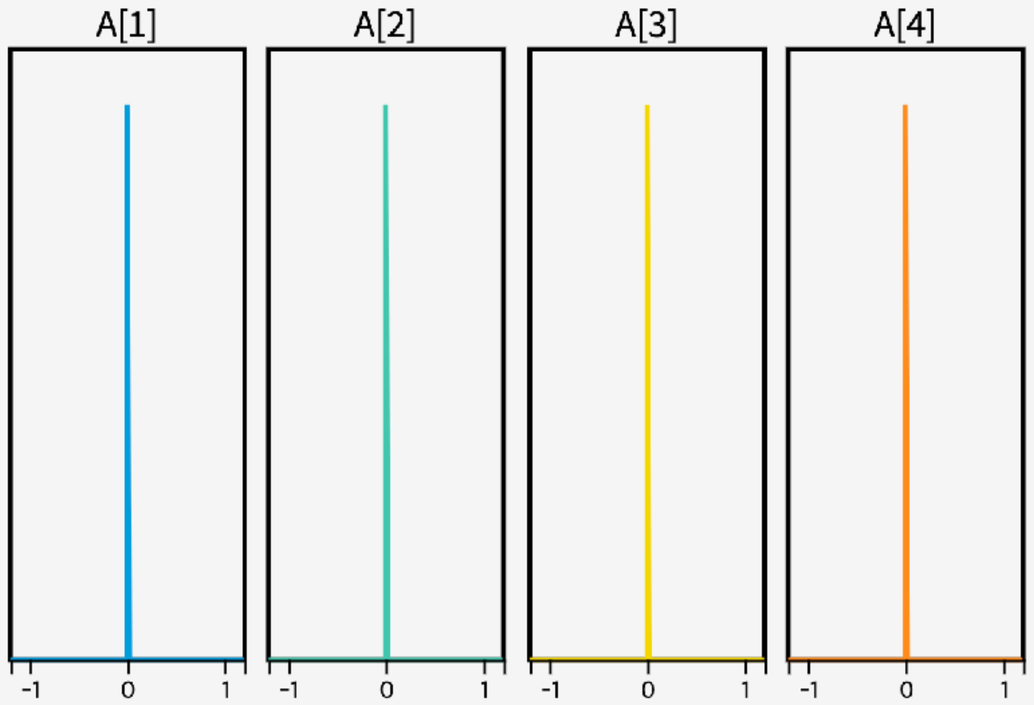
Input batch of 100 images

Batch: 6 Epoch: 0

2. Select an initialization method

Among the below distributions, select the one to use to initialize your parameters[3].

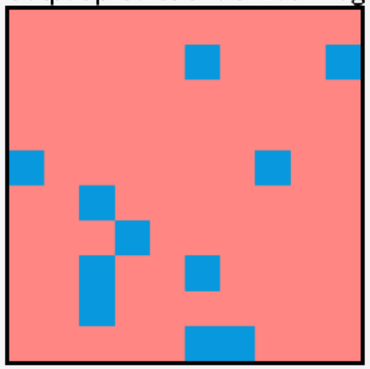
(•) Zero ( ) Uniform ( ) Xavier ( ) Standard Normal



3. Train the network and observe

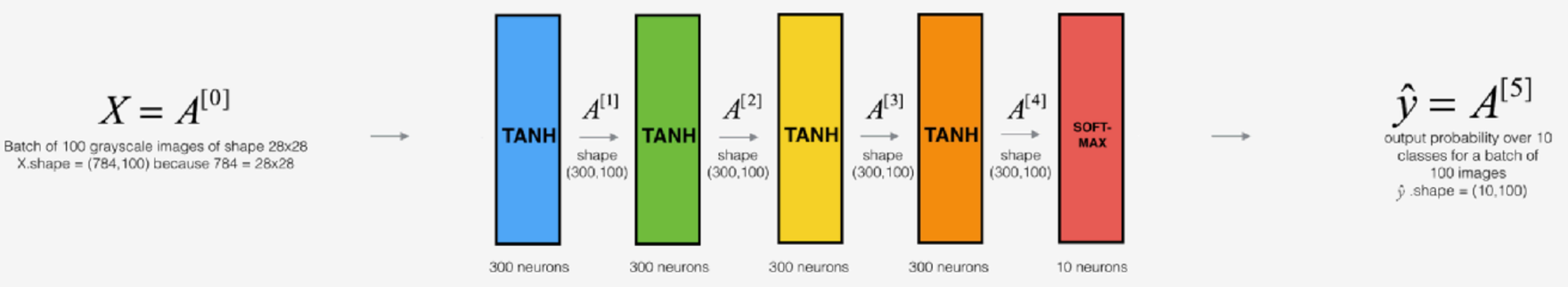
The grid below refers to the input images, Blue squares represent correctly classified images. Red squares represent misclassified images.

Output predictions of 100 images



Misclassified: 89/100 Cost: 2.30

2. Generalization initialization



1. Load your dataset

Load 10,000 handwritten digits images (MNIST).

Load MNIST (100%)

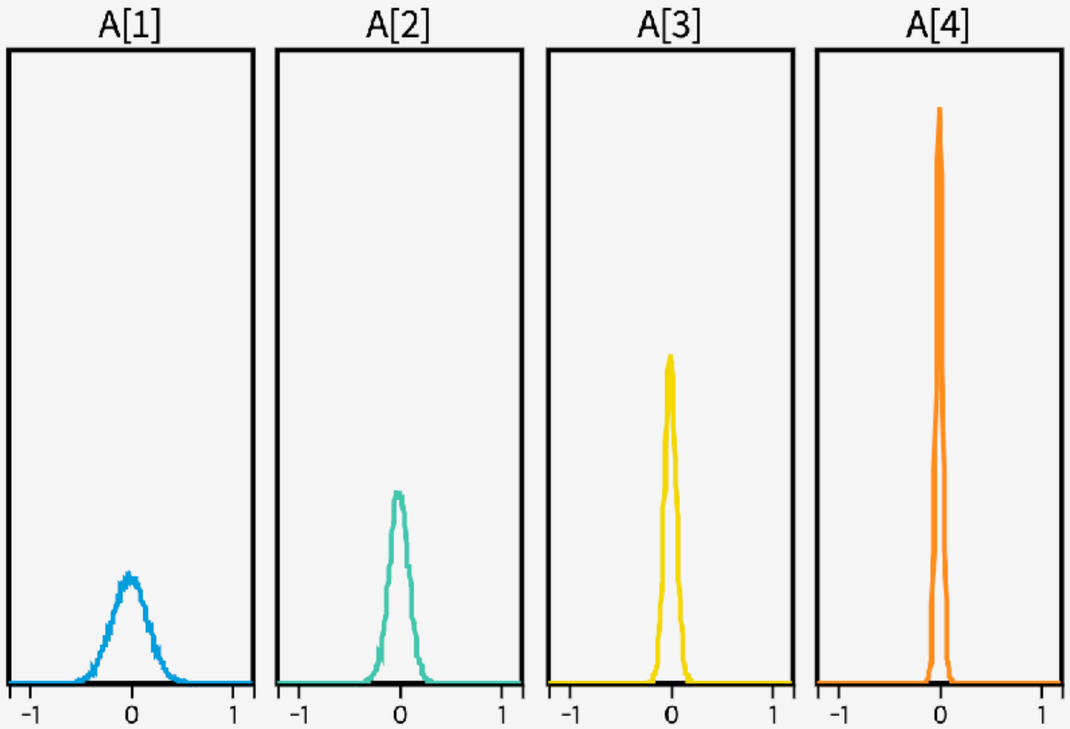
Input batch of 100 images

Batch: 3 Epoch: 0

2. Select an initialization method

Among the below distributions, select the one to use to initialize your parameters[3].

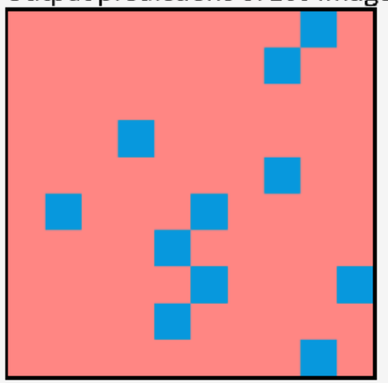
○ Zero ◉ Uniform ○ Xavier ○ Standard Normal



3. Train the network and observe

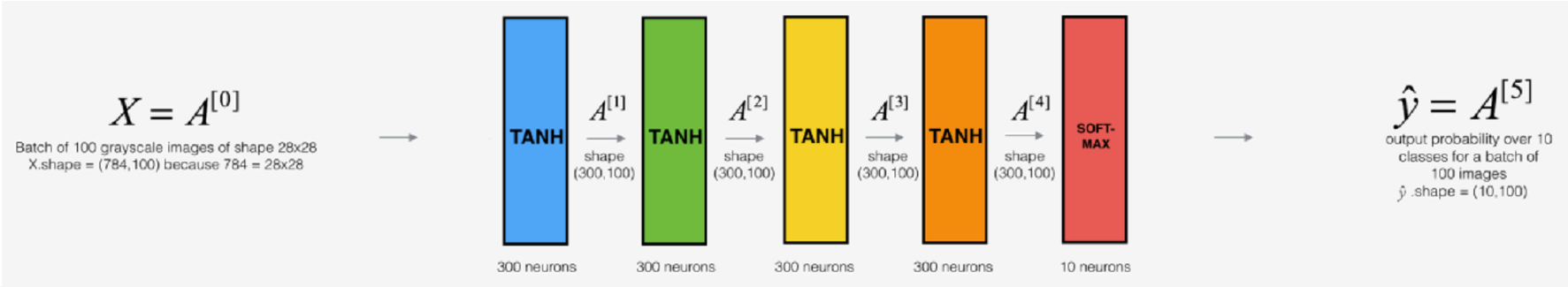
The grid below refers to the input images, Blue squares represent correctly classified images. Red squares represent misclassified images.

Output predictions of 100 images



Misclassified: 89/100 Cost: 2.31

2. Generalization initialization



1. Load your dataset

Load 10,000 handwritten digits images (MNIST).

Load MNIST (100%)

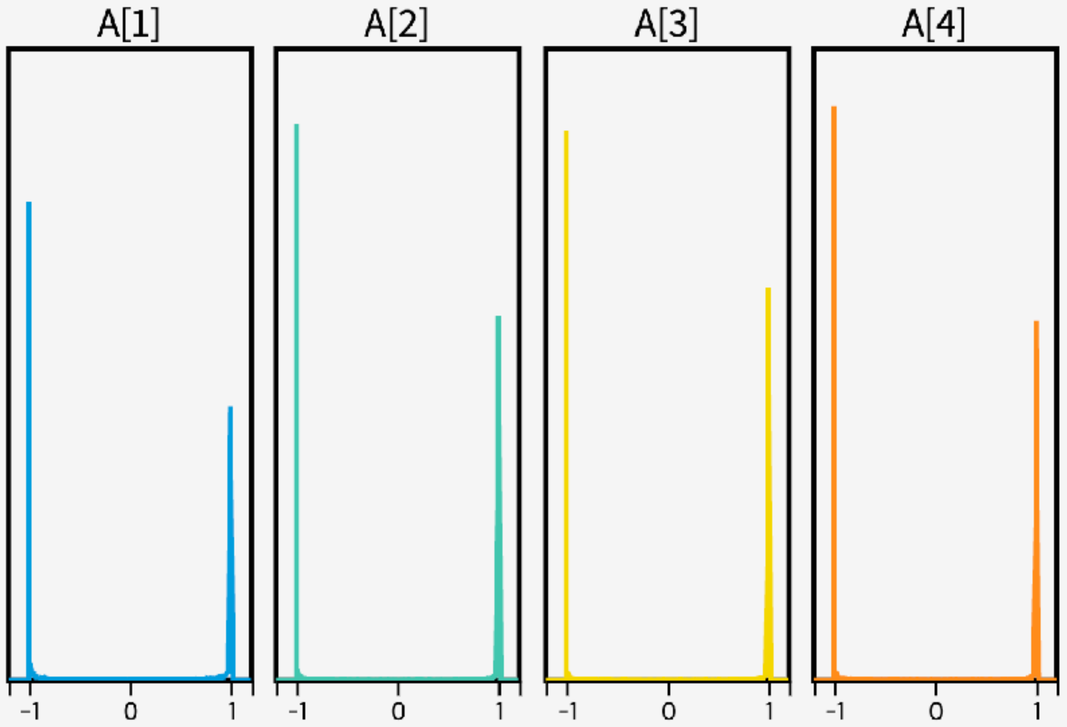
Input batch of 100 images

Batch: 11 Epoch: 0

2. Select an initialization method

Among the below distributions, select the one to use to initialize your parameters[3].

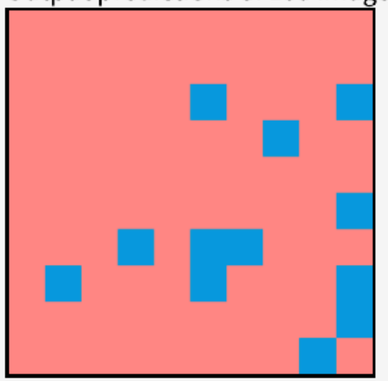
○ Zero ○ Uniform ○ Xavier ◉ Standard Normal



3. Train the network and observe

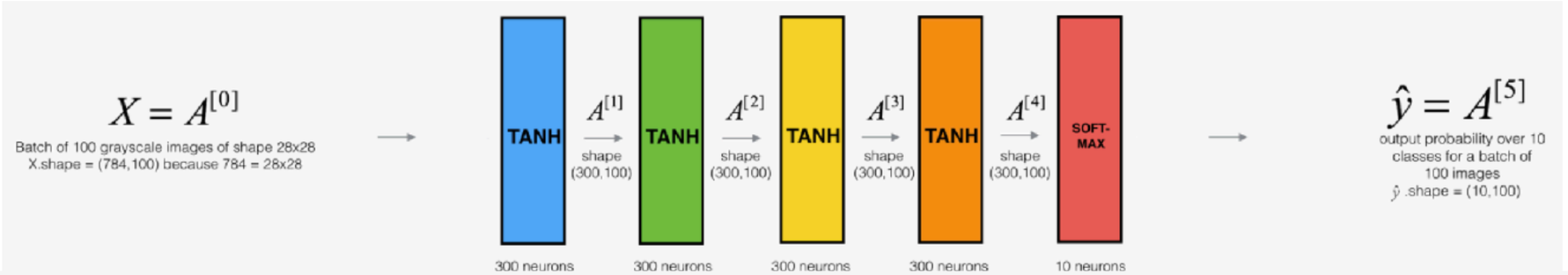
The grid below refers to the input images, Blue squares represent correctly classified images. Red squares represent misclassified images.

Output predictions of 100 images



Misclassified: 88/100 Cost: 26.92

2. Generalization initialization



1. Load your dataset

Load 10,000 handwritten digits images (MNIST).

Load MNIST (100%)

2. Select an initialization method

Among the below distributions, select the one to use to initialize your parameters[3].

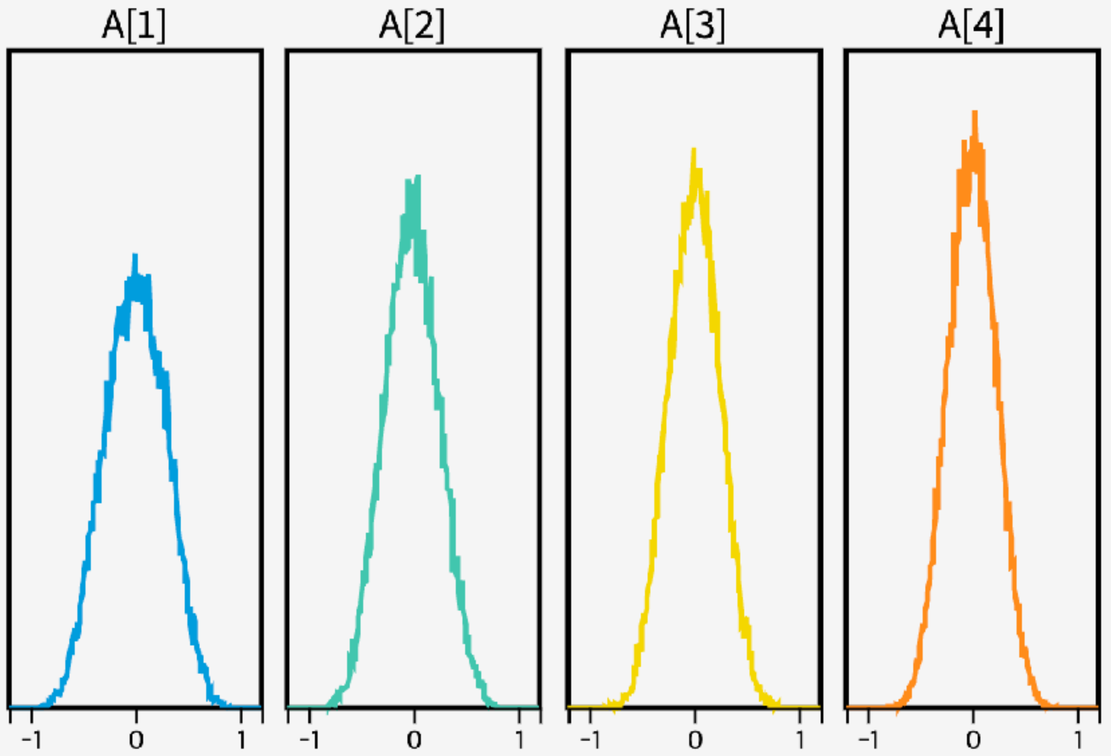
Zero Uniform Xavier Standard Normal

3. Train the network and observe

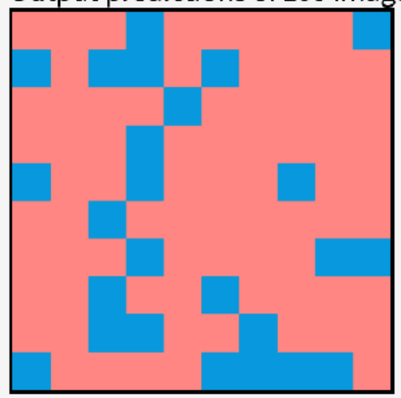
The grid below refers to the input images, Blue squares represent correctly classified images. Red squares represent misclassified images.

Input batch of 100 images

Batch: 6 Epoch: 0



Output predictions of 100 images



Misclassified: 75/100 Cost: 2.22

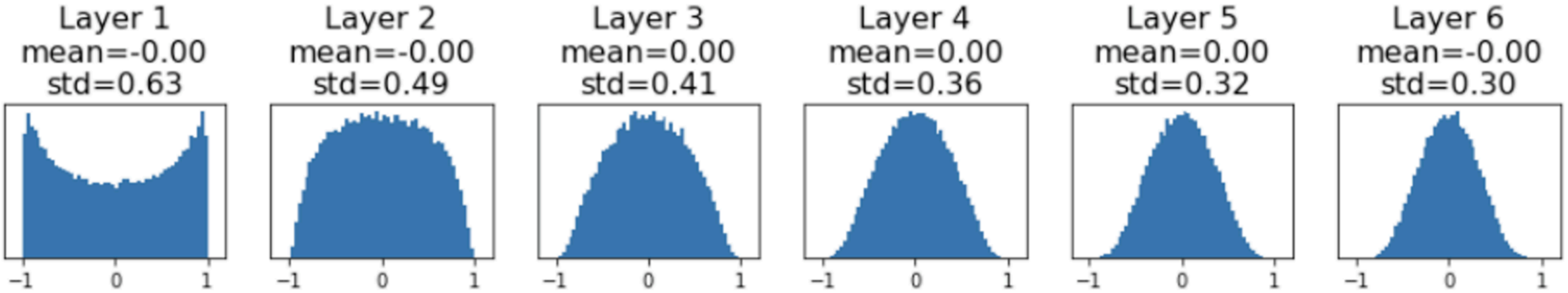
Weight Initialization: Xavier Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is kernel_size^2 * input_channels



Weight Initialization: Xavier Initialization

"Xavier" initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

$y = Wx$

$$y_i = \sum_{j=1}^{Din} x_j w_j$$

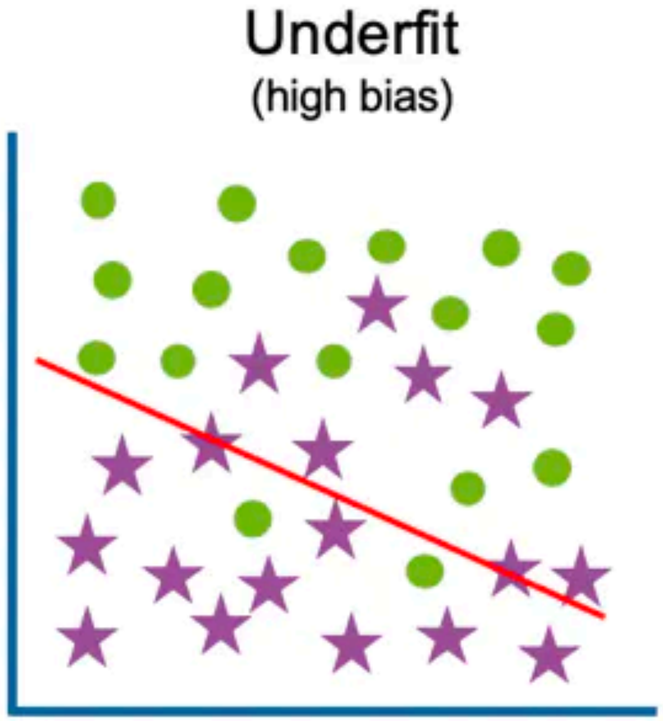
$Var(y_i) = Din * Var(x_i w_i)$ [Assume x, w are iid]

$= Din * (E[x_i^2]E[w_i^2] - E[x_i]^2 E[w_i]^2)$ [Assume x, w independent]

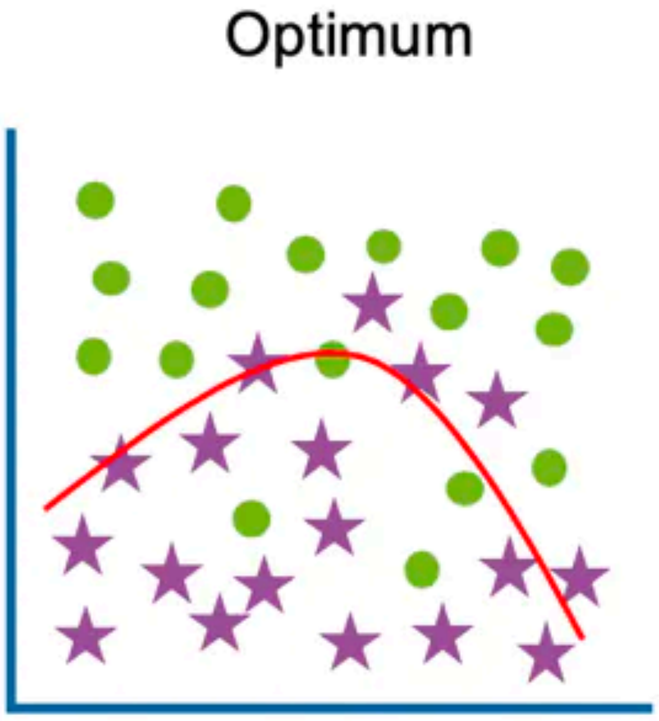
$= Din * Var(x_i) * Var(w_i)$ [Assume x, w are zero-mean]

If $Var(w_i) = 1/Din$ then $Var(y_i) = Var(x_i)$

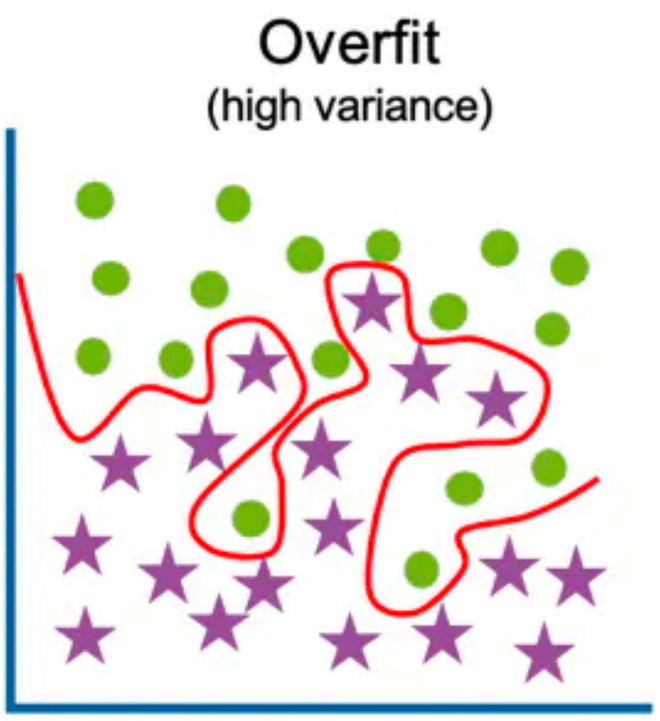
2. Generalization
Overfitting



High training error
High test error



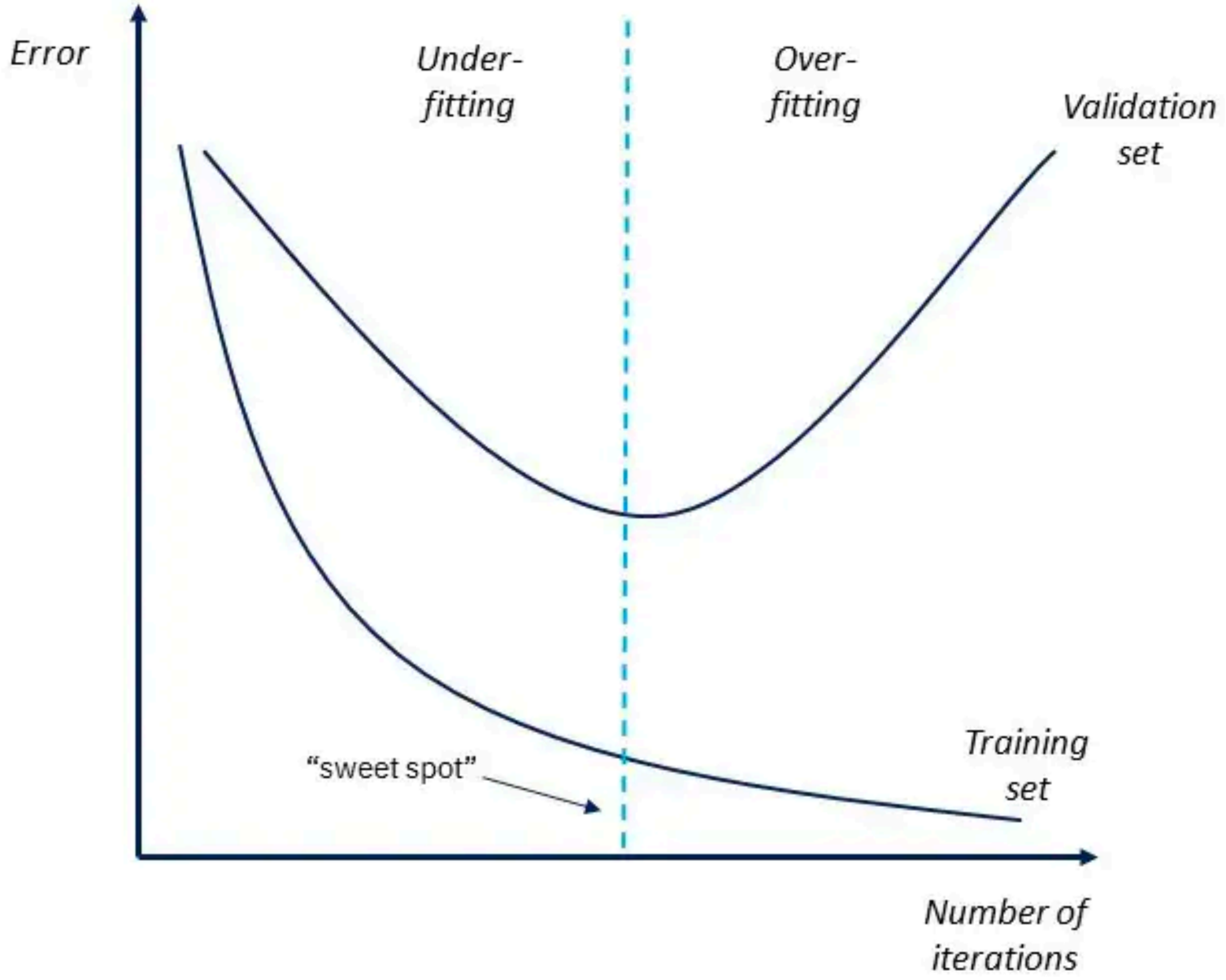
Low training error
Low test error



Low training error
High test error

https://www.ibm.com/cloud/learn/overfitting

2. Generalization
Overfitting



https://www.ibm.com/cloud/learn/overfitting

2. Generalization
Early Stopping

```
>>> callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)
>>> # This callback will stop the training when there is no improvement in
>>> # the loss for three consecutive epochs.
>>> model = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])
>>> model.compile(tf.keras.optimizers.SGD(), loss='mse')
>>> history = model.fit(np.arange(100).reshape(5, 20), np.zeros(5),
...                     epochs=10, batch_size=1, callbacks=[callback],
...                     verbose=0)
>>> len(history.history['loss'])  # Only 4 epochs are run.
4
```

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

λ is a hyperparameter giving regularization strength

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

L1 Regularization(Lasso)

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n}|\theta_j|$$

L2 Regularization(Ridge)

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

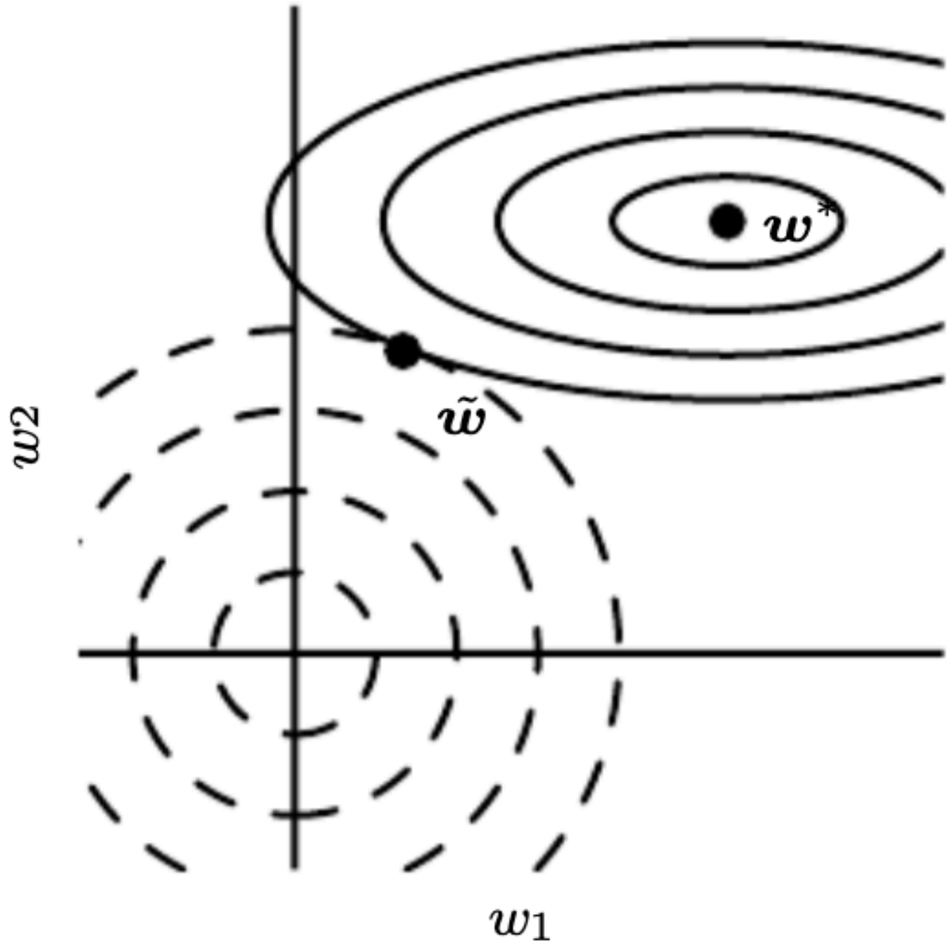
$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n}\theta_j^2$$

L2 regularization prefers weights to be "spread out"

$$w_1^T x = w_2^T x = 1$$

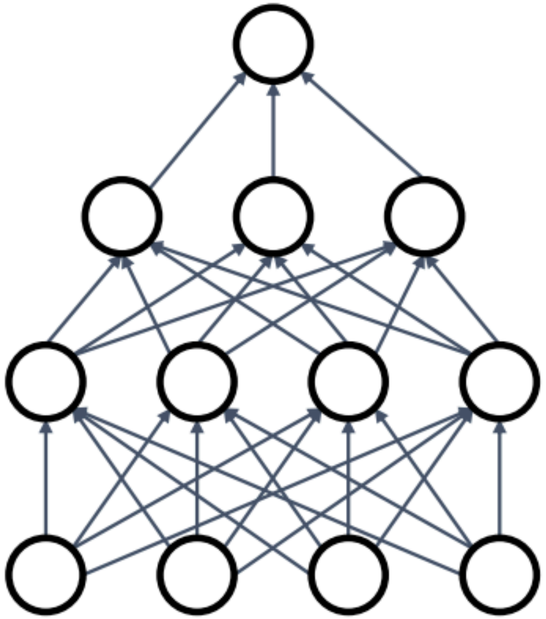
Same predictions, so data loss will always be the same

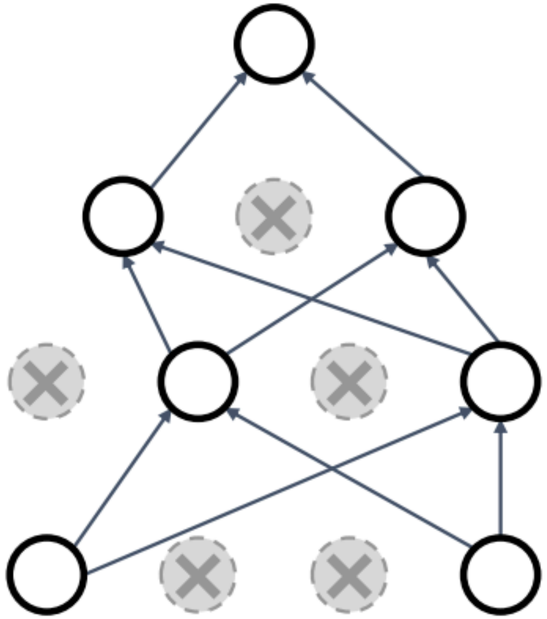
L2 Regularization(Ridge)



Regularization: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common





2. Generalization
Dropout

Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

Dropout: Test Time

Dropout makes our output random!

Output (label) Input (image)

$$y = f_W(x, z)$$

Random mask

Want to "average out" the randomness at test-time

$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

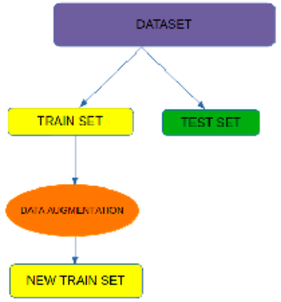
2. Generalization
Dropout

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

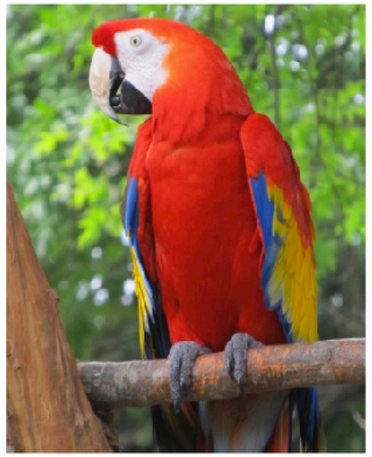
Inverted dropout

```
def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3
```

Image Data Augmentation

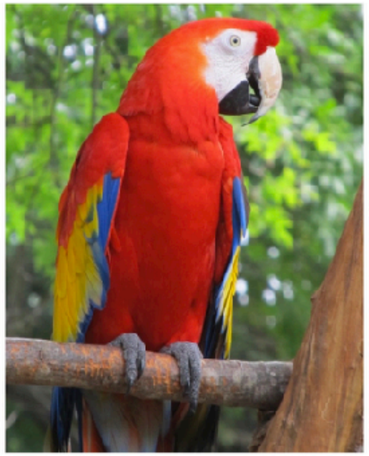


Original image

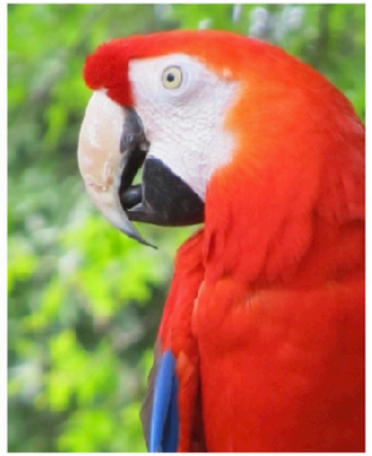


augmentation

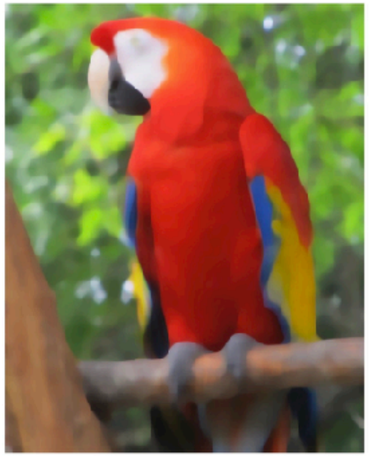
Horizontal Flip



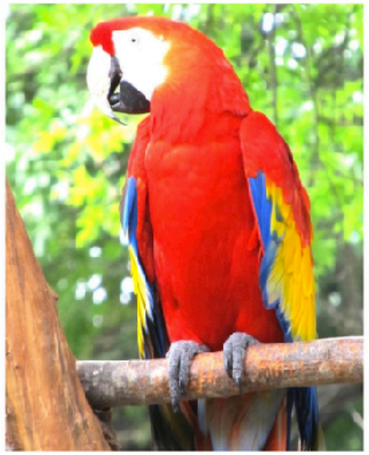
Crop



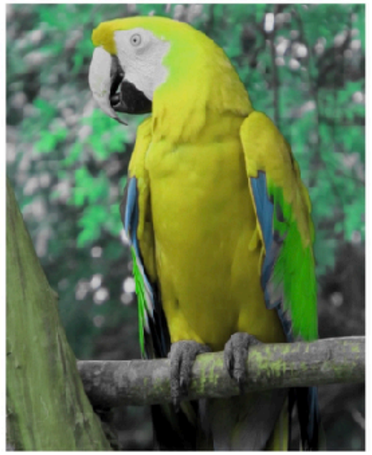
Median Blur



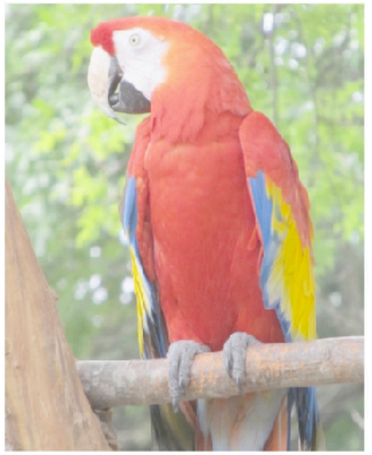
Contrast



Hue / Saturation / Value



Gamma

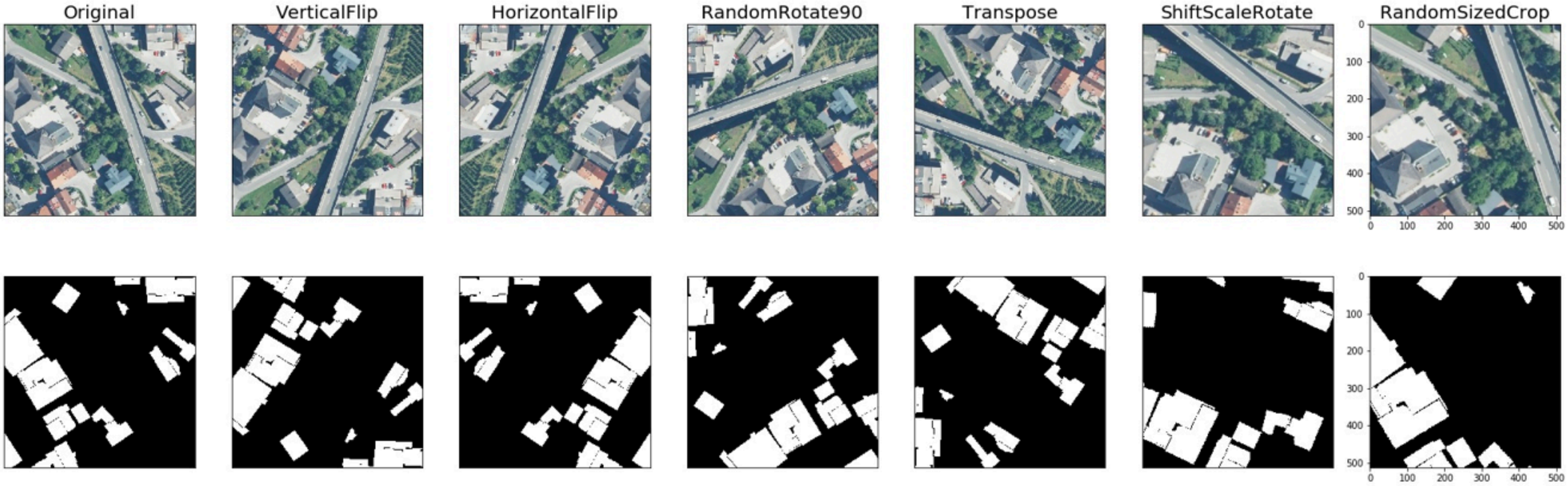


https://albumentations.ai/docs/introduction/image_augmentation/

Pixel-Level Transforms

https://hoya012.github.io/blog/Image-Data-Augmentation-Overview/

Spatial-Level Transforms



https://hoya012.github.io/blog/Image-Data-Augmentation-Overview/

PatchShuffle regularization

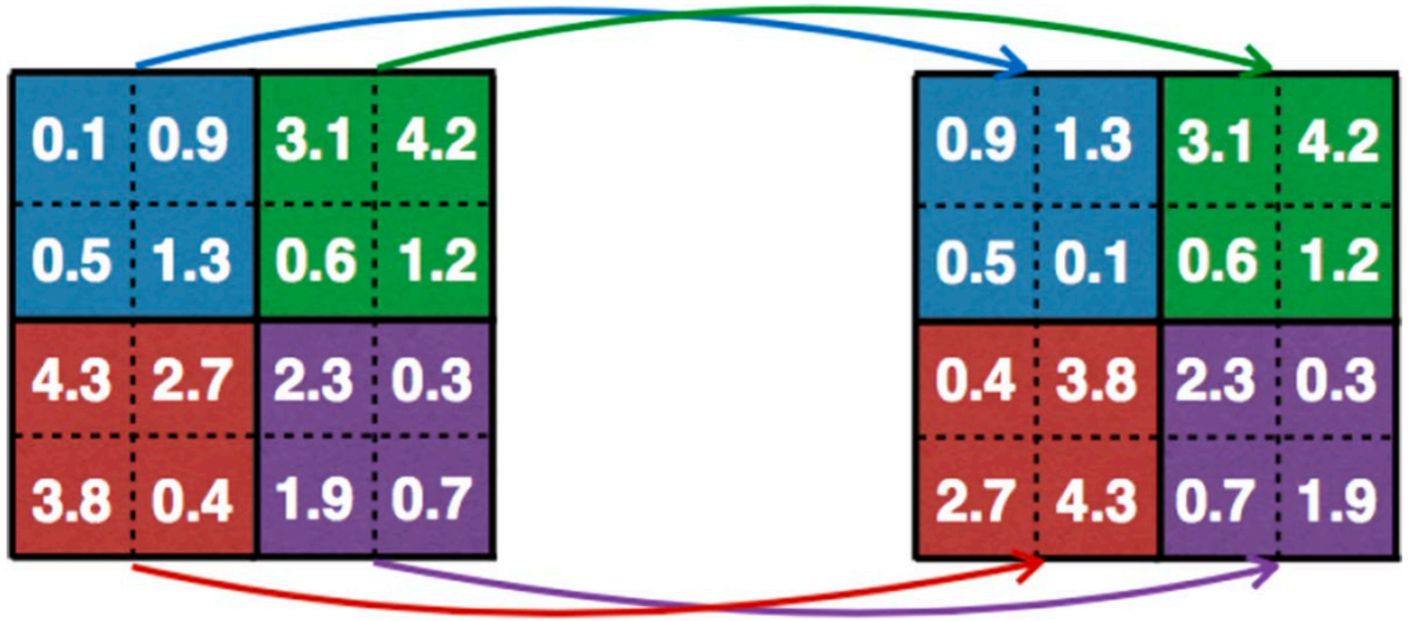


Fig. 5 Examples of applying the PatchShuffle regularization technique [64]

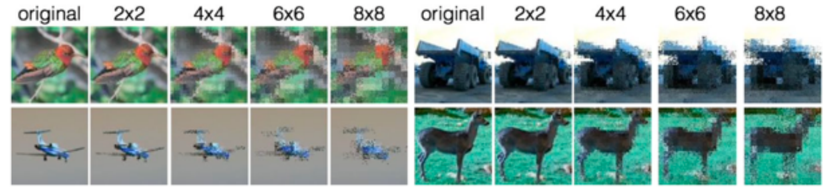


Fig. 6 Pixels in a n × n window are randomly shifted with a probability parameter p

 Guoliang K, Xuanyi D, Liang Z, Yi Y. PatchShuffle regularization. arXiv preprint. 2017.

PatchShuffle regularization

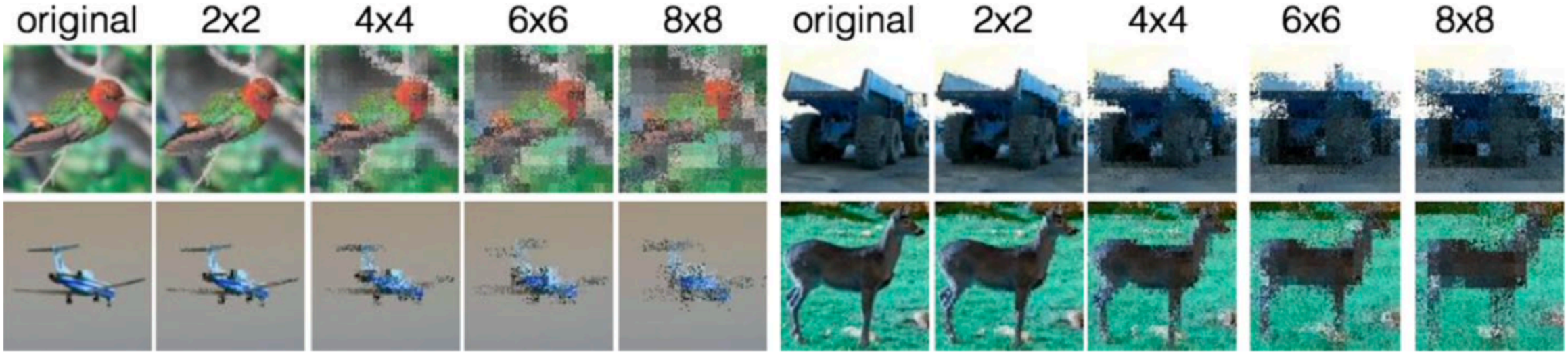
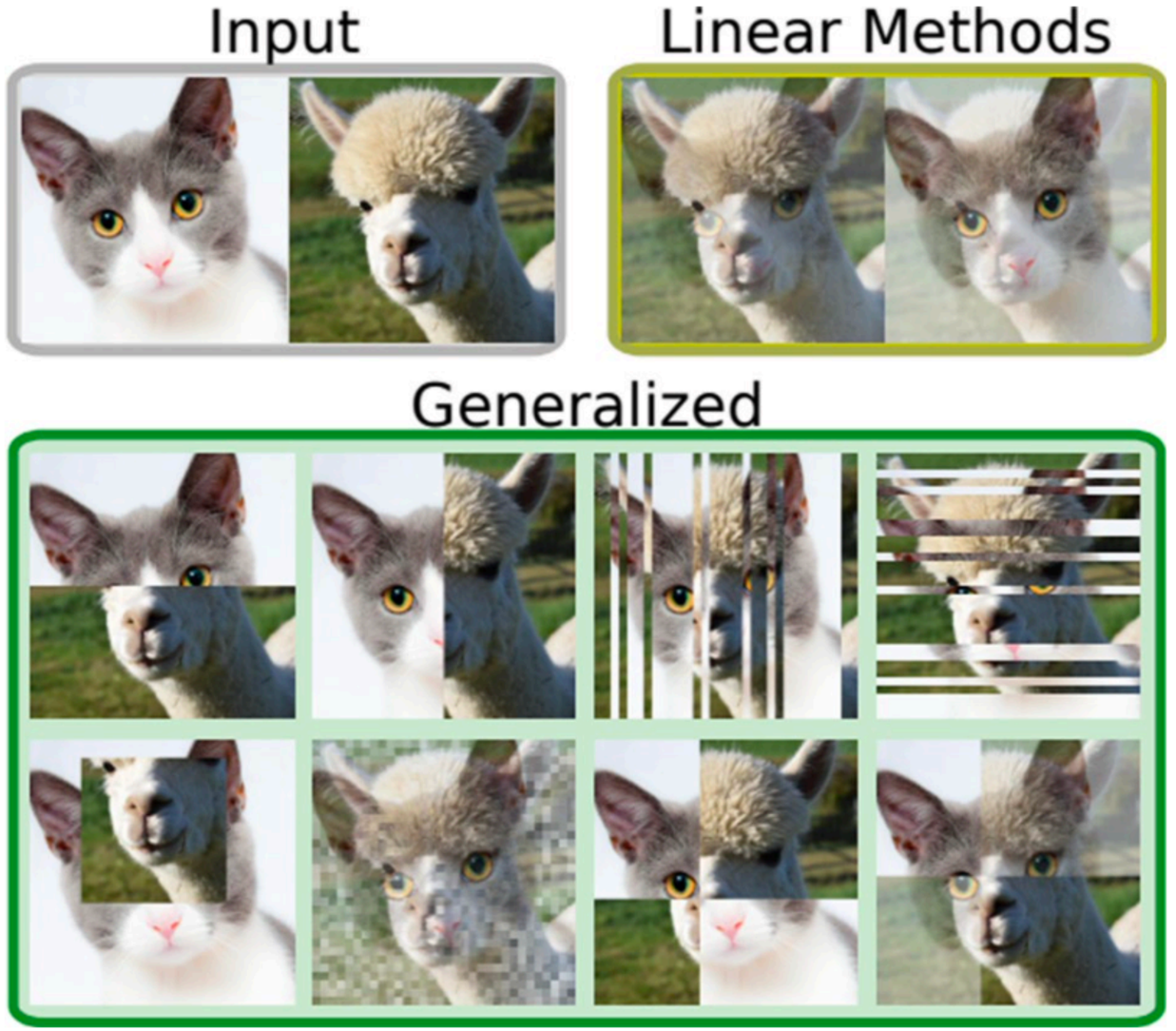


Fig. 6 Pixels in a n × n window are randomly shifted with a probability parameter p

 Guoliang K, Xuanyi D, Liang Z, Yi Y. PatchShuffle regularization. arXiv preprint. 2017.

Non-linearly mixing images



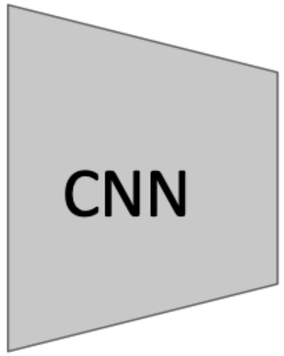
Cecilia S, Michael JD. Improved mixed-example data augmentation. ArXiv preprint. 2018.

Cutout

Figure 1: Cutout applied to images from the CIFAR-10 dataset.

mixup

$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j, \qquad \text{where } x_i, x_j \text{ are raw input vectors}$$
$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j, \qquad \text{where } y_i, y_j \text{ are one-hot label encodings}$$



Target label:
cat: 0.4
dog: 0.6

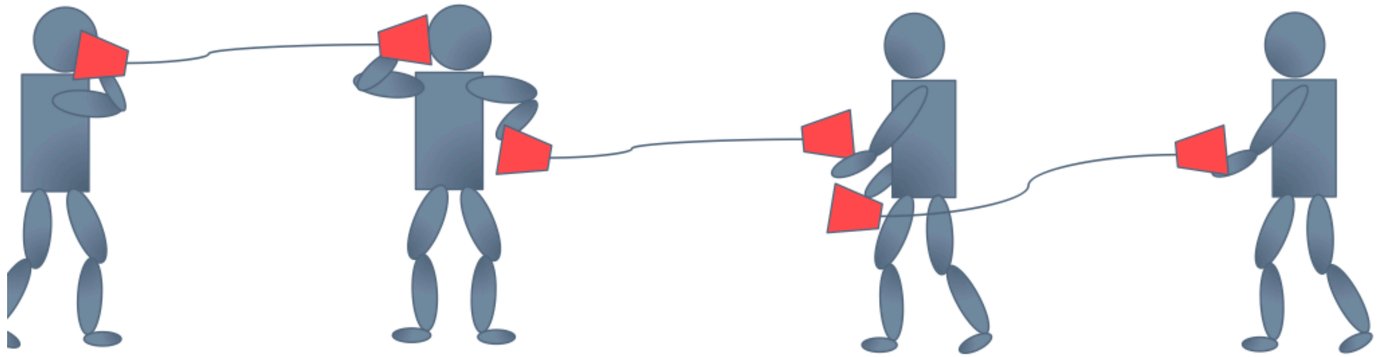
Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

mixup: BEYOND EMPIRICAL RISK MINIMIZATION

NLP에는 Augmentation이 없을까?

1. Back Translation
2. Easy Data Augmentation
   - Synonym Replacement
   - Random Insertion
   - Random Swap
   - Random Deletion
3. Applying Data Augmentation skill for CV to NLP

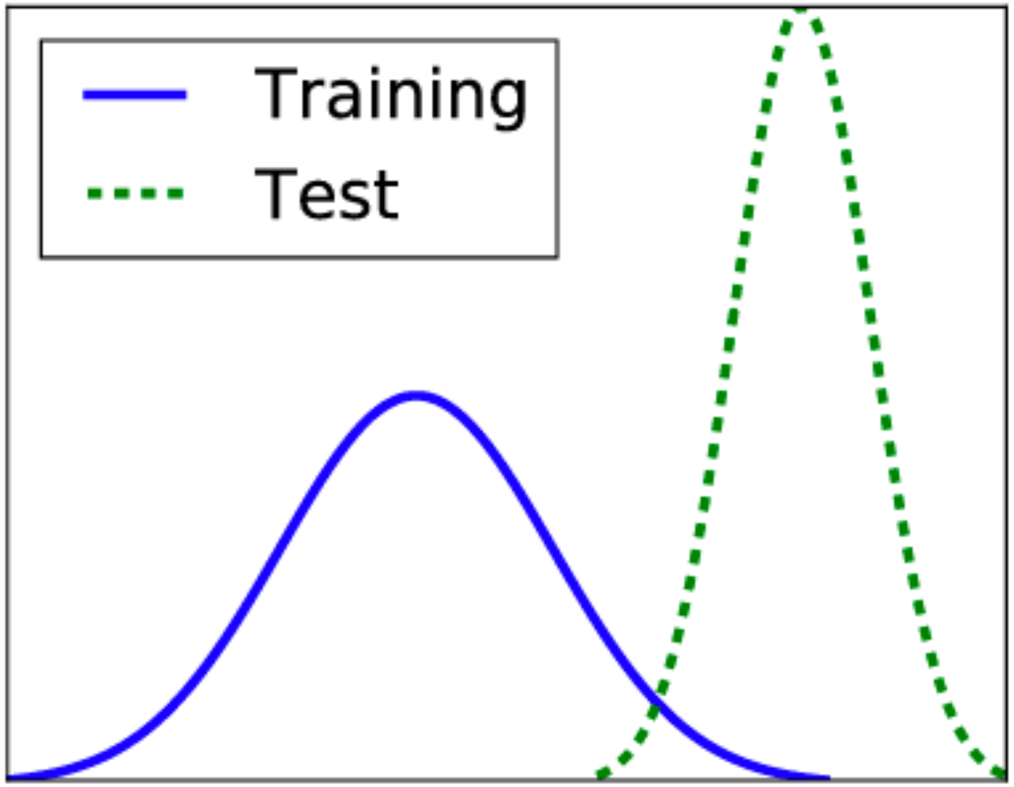
Internal Covariate Shift



Batch Normalization – What the hey?

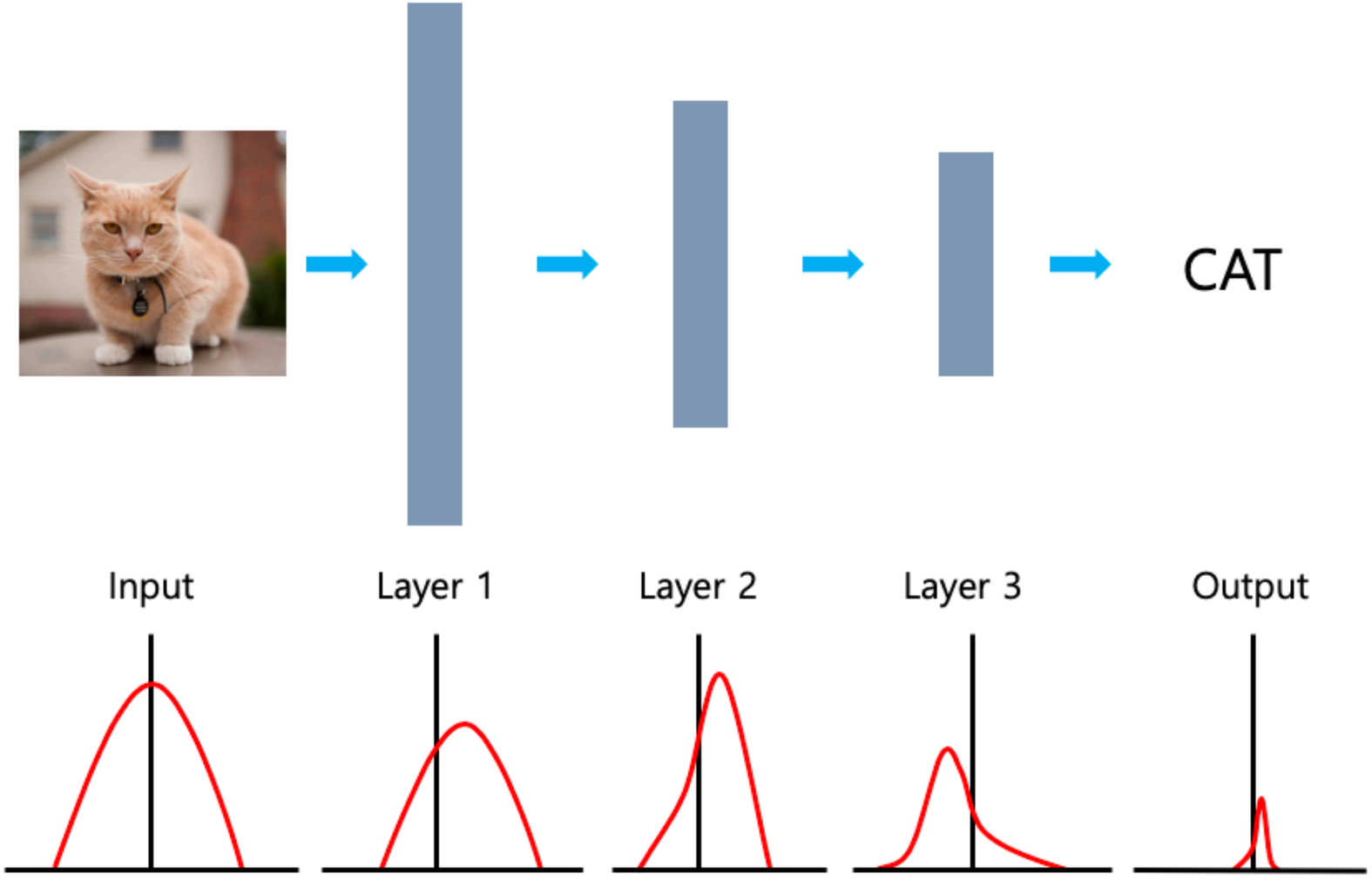
Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift

Covariate Shift



On the Reduction of Biases in Big Data Sets for the Detection of Irregular Power Usage
Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift

Internal Covariate Shift



Normalization via Mini-Batch Statistics

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Input: Network N with trainable parameters Θ;
subset of activations $\{x^{(k)}\}_{k=1}^{K}$
Output: Batch-normalized network for inference, $N_{\mathrm{BN}}^{\mathrm{inf}}$

1: $N_{\mathrm{BN}}^{\mathrm{tr}} \leftarrow N$ // Training BN network
2: **for** $k = 1 \ldots K$ **do**
3: Add transformation $y^{(k)} = \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\mathrm{BN}}^{\mathrm{tr}}$ (Alg. 1)
4: Modify each layer in $N_{\mathrm{BN}}^{\mathrm{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
5: **end for**
6: Train $N_{\mathrm{BN}}^{\mathrm{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
7: $N_{\mathrm{BN}}^{\mathrm{inf}} \leftarrow N_{\mathrm{BN}}^{\mathrm{tr}}$ // Inference BN network with frozen // parameters
8: **for** $k = 1 \ldots K$ **do**
9: // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10: Process multiple training mini-batches $\mathcal{B}$, each of size m, and average over them:

$$\mathrm{E}[x] \leftarrow \mathrm{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$
$$\mathrm{Var}[x] \leftarrow \frac{m}{m-1}\mathrm{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

11: In $N_{\mathrm{BN}}^{\mathrm{inf}}$, replace the transform $y = \mathrm{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$
12: **end for**

Algorithm 2: Training a Batch-Normalized Network

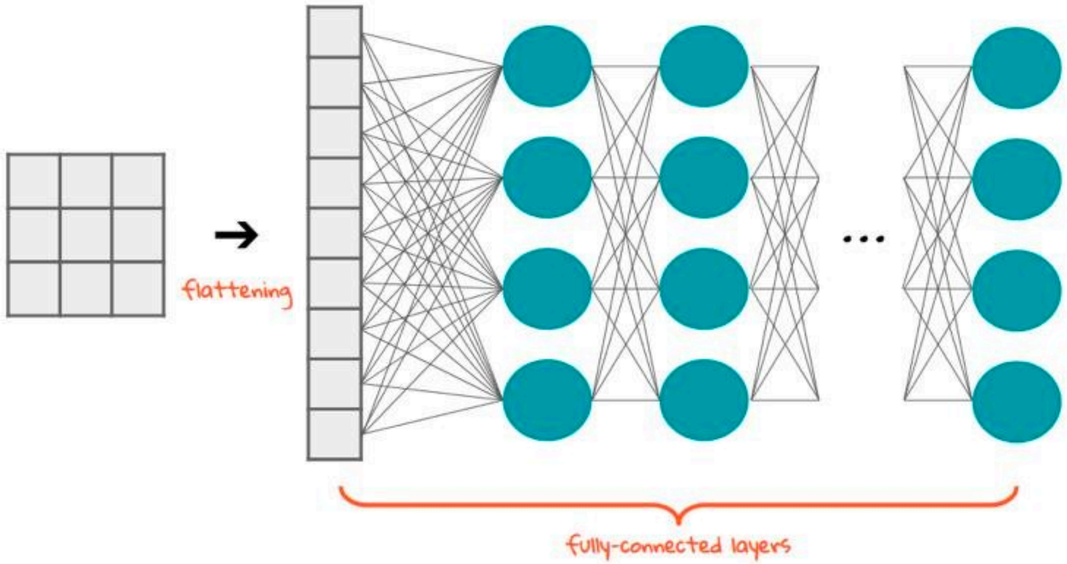
tf.keras.layers.BatchNormalization

During inference (i.e. when using `evaluate()` or `predict()` or when calling the layer/model with the argument `training=False` (which is the default), the layer normalizes its output using a moving average of the mean and standard deviation of the batches it has seen during training. That is to say, it returns `gamma * (batch - self.moving_mean) / sqrt(self.moving_var + epsilon) + beta`.

`self.moving_mean` and `self.moving_var` are non-trainable variables that are updated each time the layer in called in training mode, as such:

- `moving_mean = moving_mean * momentum + mean(batch) * (1 - momentum)`
- `moving_var = moving_var * momentum + var(batch) * (1 - momentum)`

실습

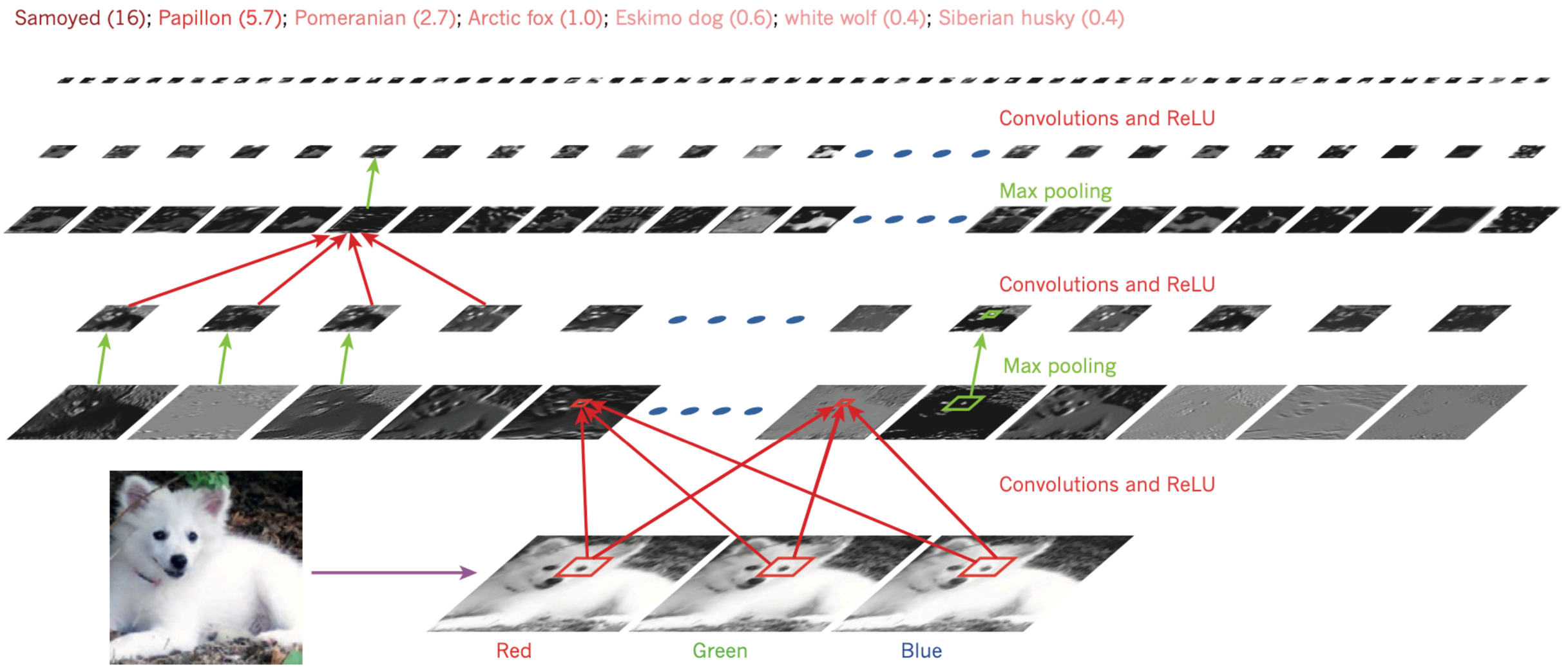


1. Image의 공간적 특성을 학습에 이용하지 못 함
2. data의 크기가 커짐에 따라 paramters의 수가 매우 커짐

 https://www.researchgate.net/figure/Flattening-step-in-CNN_fig2_343263135

Motivation

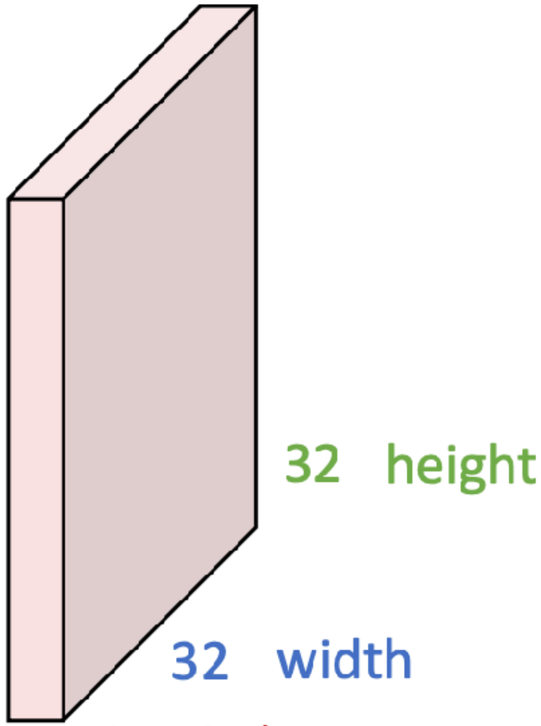
1. Sparse interactions
2. Parameter sharing
3. Equivariant representations


Samoyed (16); Papillon (5.7); Pomeranian (2.7); Arctic fox (1.0); Eskimo dog (0.6); white wolf (0.4); Siberian husky (0.4)
Convolutions and ReLU
Max pooling
Convolutions and ReLU
Max pooling
Convolutions and ReLU
Red
Green
Blue

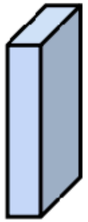
Convolution Layer

Filters always extend the full depth of the input volume

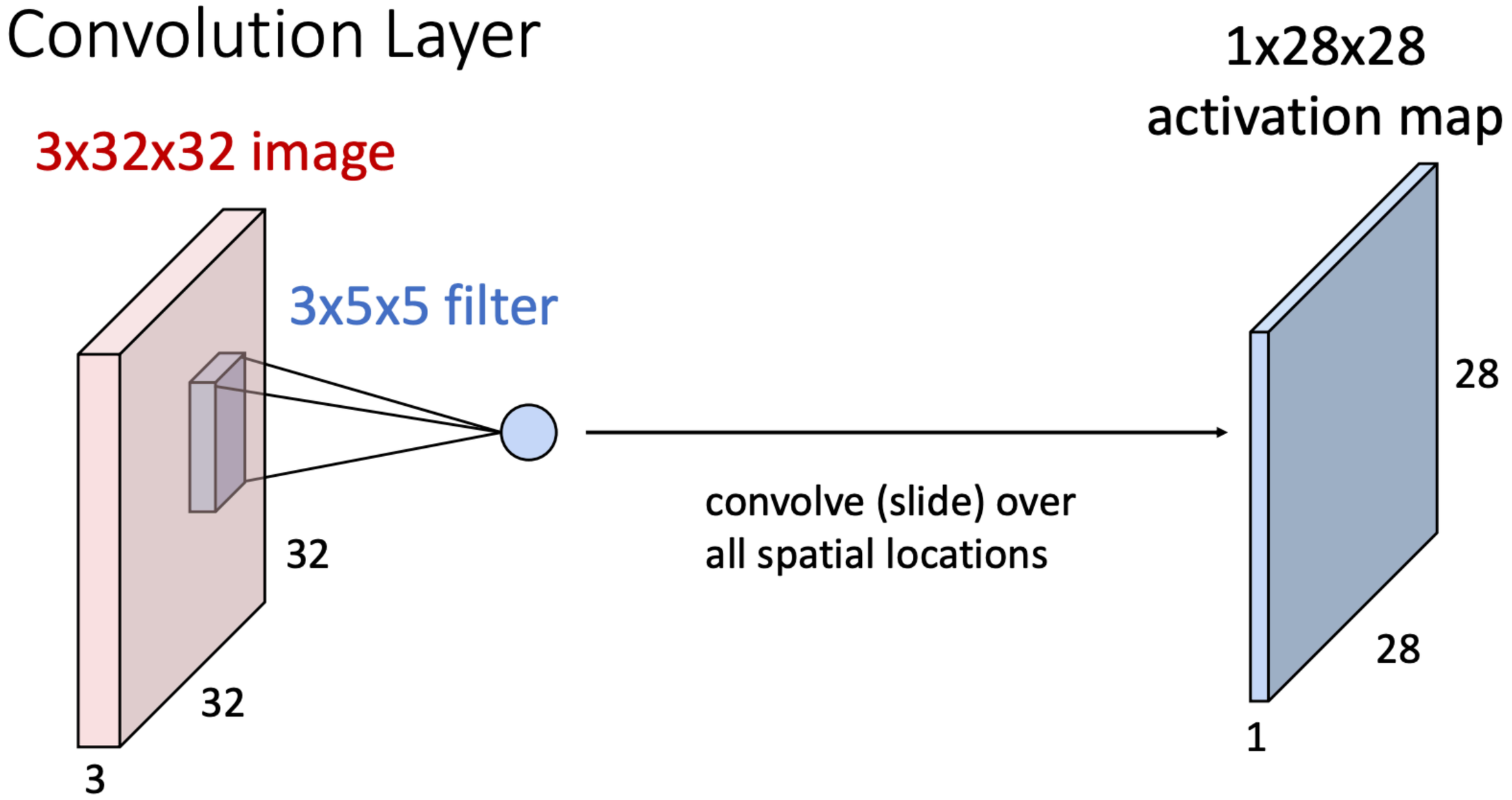
3x32x32 image



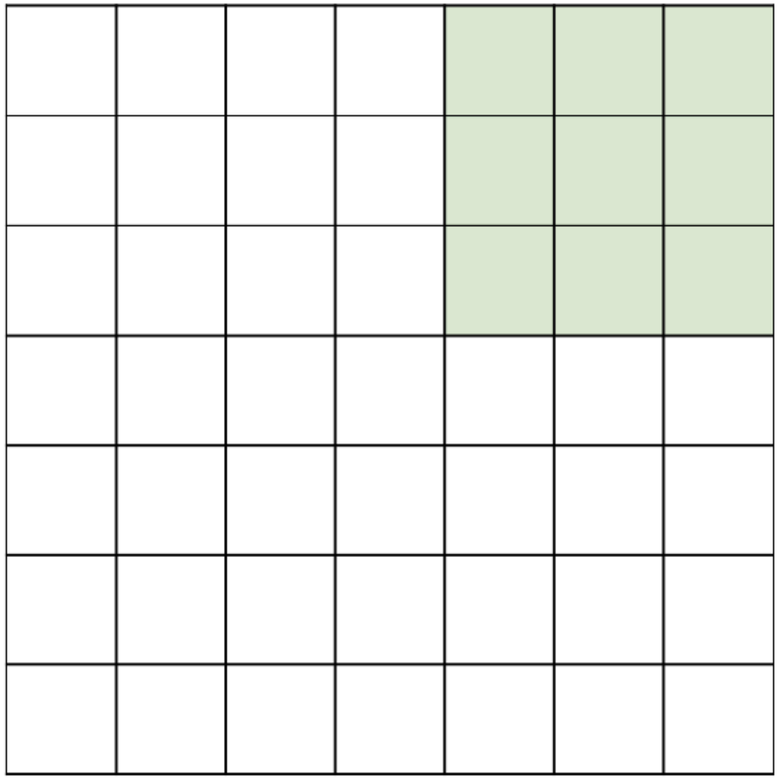
3x5x5 filter



Convolve the filter with the image
i.e. “slide over the image spatially, computing dot products”


Convolution Layer
3x32x32 image
3x5x5 filter
32
32
3
convolve (slide) over
all spatial locations
1x28x28
activation map
28
28
1

Stride



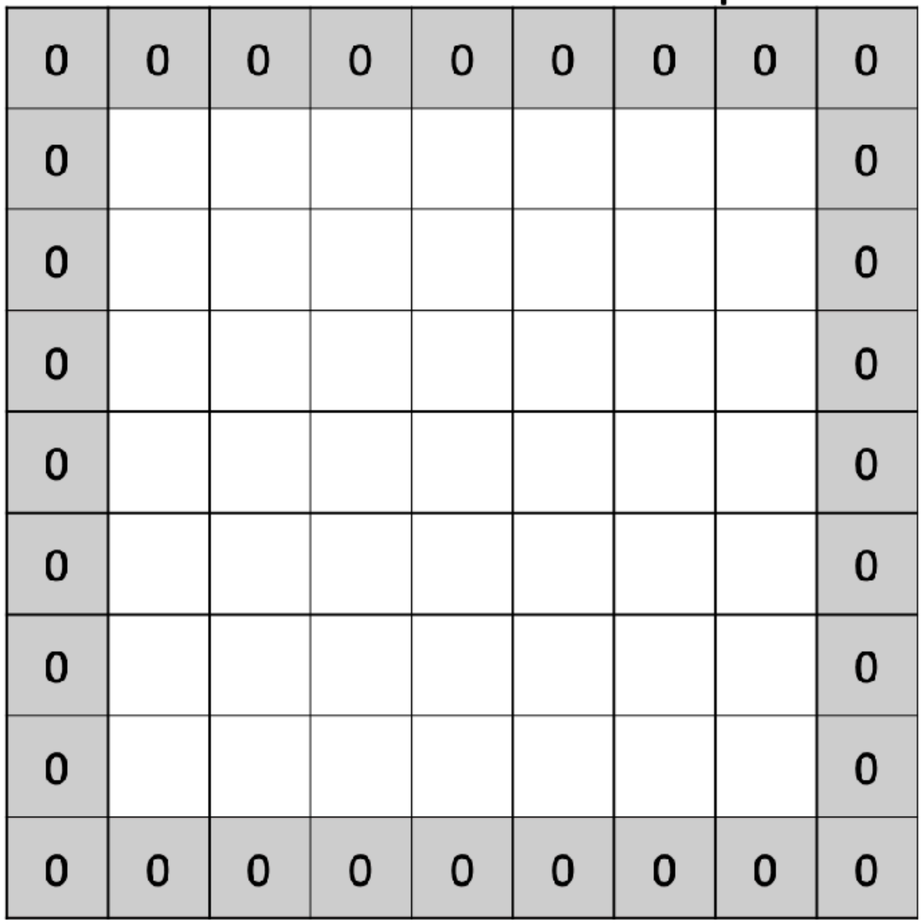
Input: 7x7
Filter: 3x3
Stride: 2

Output: 3x3

In general:
Input: W
Filter: K
Padding: P
Stride: S
Output: (W – K + 2P) / S + 1

Padding

A closer look at spatial dimensions



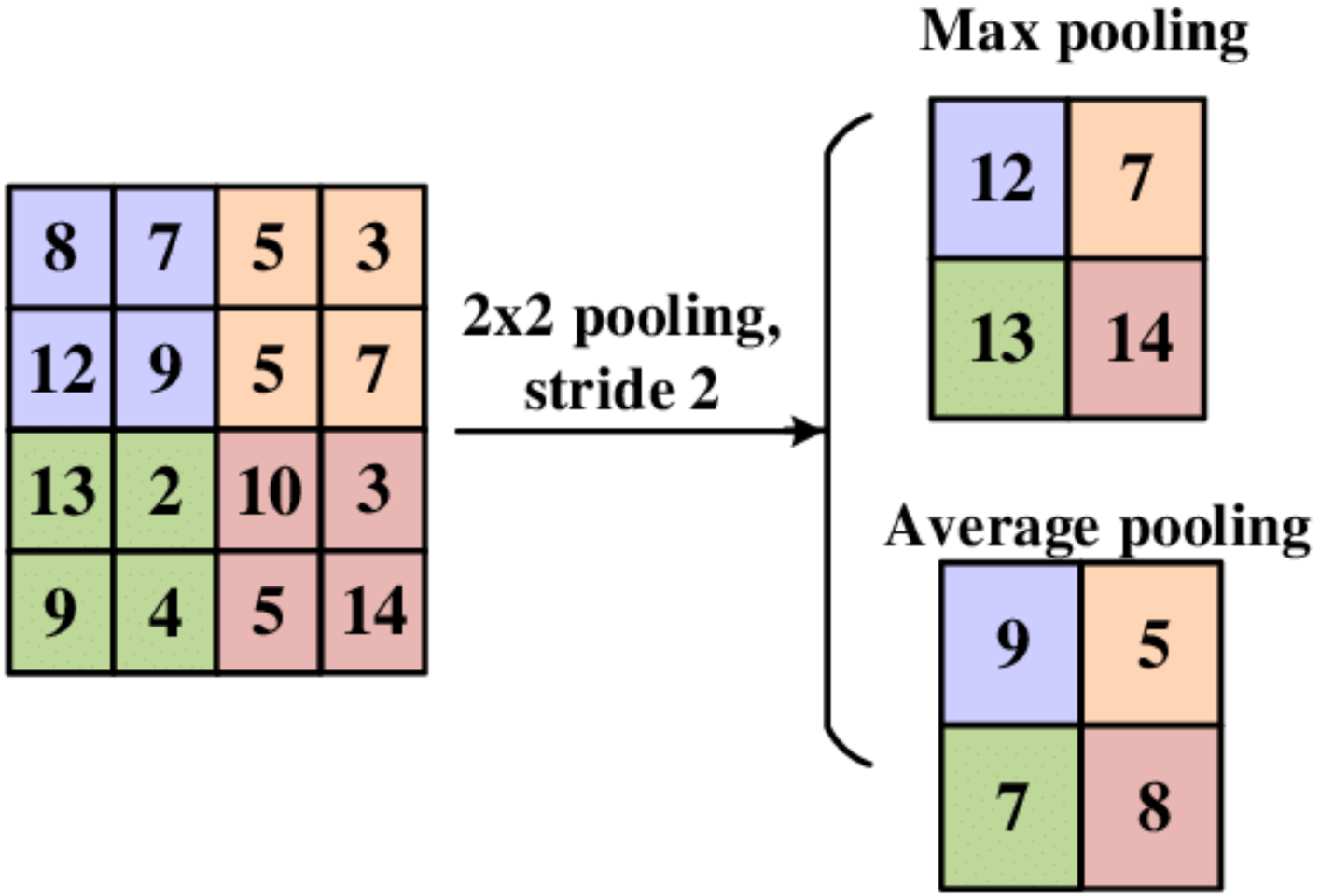
Input: 7x7
Filter: 3x3
Output: 5x5

In general:
Input: W
Filter: K
Output: W – K + 1

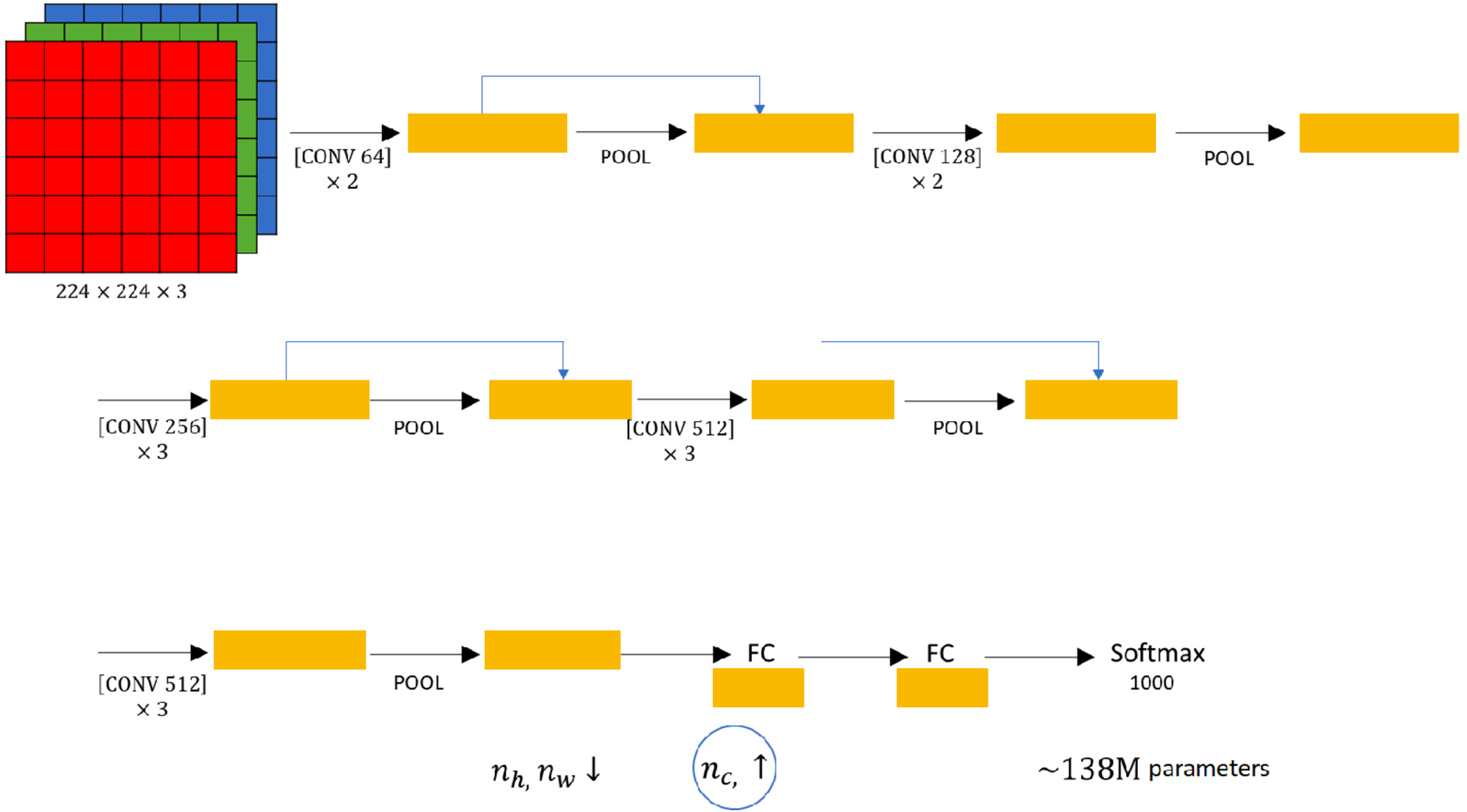
Problem: Feature maps "shrink" with each layer!

Solution: **padding**
Add zeros around the input

Pooling


8 7 5 3
12 9 5 7
13 2 10 3
9 4 5 14
2x2 pooling,
stride 2
Max pooling
12 7
13 14
Average pooling
9 5
7 8

CONV = 3 x 3 filter, s=1, same MAX-POOL = 2 x 2, s=2



실습