Stanford Univ., CS231n – Computer Vision

2022. 08. 18

Seonghak KIM
Electrical Engineering, EE

Image Classification

Given image → discrete labels (classes)

- Image is expressed by the numbers between [0 ,255] with 3 channels RGB
- Challenges
  - Viewpoint variation: pixels change according to the camera location (not changed image)
  - Illumination
  - Deformation
  - Occlusion: the hided object
  - Background clutter: background has a similar color with the object.
  - Intraclass variation: the objects that has the same class (labels)

Non-Parametric Approach

- After training a *classifier* from a dataset of images and labels, apply to the test images
- *Classifier:* Nearest Neighbor
  - L1 distance: the absolute value of pixel gap between the test image and training image

→ $d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$

[test image pixel]	[training image pixel]	[pixel-wise absolute value differences]
$\begin{matrix} 56 & 32 & 10 \\ 90 & 23 & 128 \\ 24 & 26 & 178 \end{matrix}$	$\begin{matrix} 10 & 20 & 24 \\ 8 & 10 & 89 \\ 12 & 16 & 178 \end{matrix}$	$\begin{matrix} 46 & 12 & 14 \\ 82 & 13 & 39 \\ 12 & 10 & 0 \end{matrix}$

$$\boxed{\sum a_{ij} = 228}$$

Image Classification

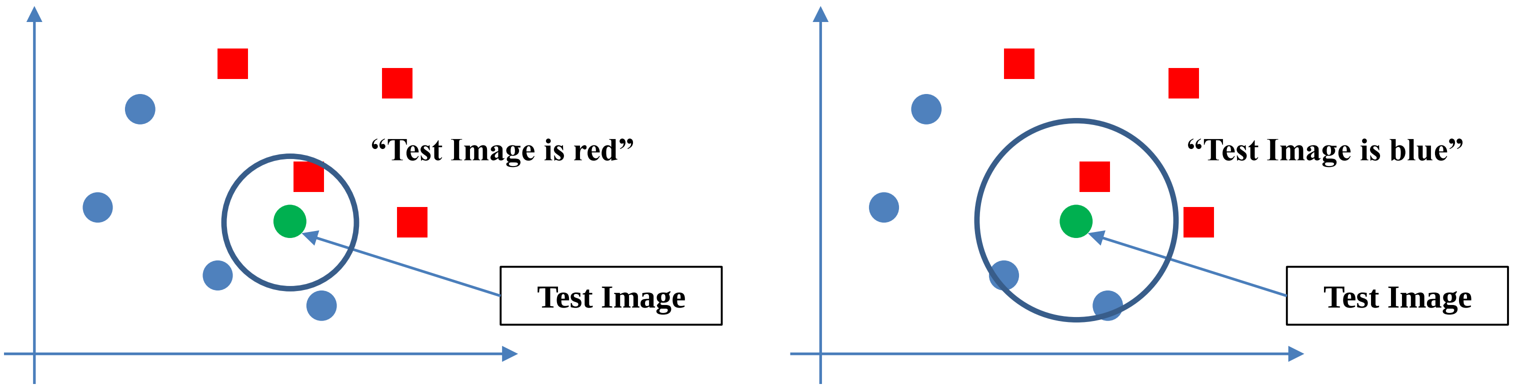
Non-Parametric Approach

- *Classifier:* K-Nearest Neighbors (KNN)
  - Majority vote from K closest points, not from nearest neighbor (i.e., Nearest Neighbors that is the case of K=1)
  - L1 distance
  - L2 distance → $d_2(I_1,\ I_2) = \sqrt{\sum_p \left(I_1^p - I_2^p\right)^2}$

→ where the best value of K and the best distance metric (i.e., L1 distance or L2 distance) are *hyperparameters,* thus it is problem-dependent.

  - Weak point: Short training time, but long prediction time; it is desirable that fast at prediction and slow for training

∴ *Convolution Neural Network (CNN)*



[Nearest Neighbors (K=1)] [K-Nearest Neighbors (K=3)]

Image Classification

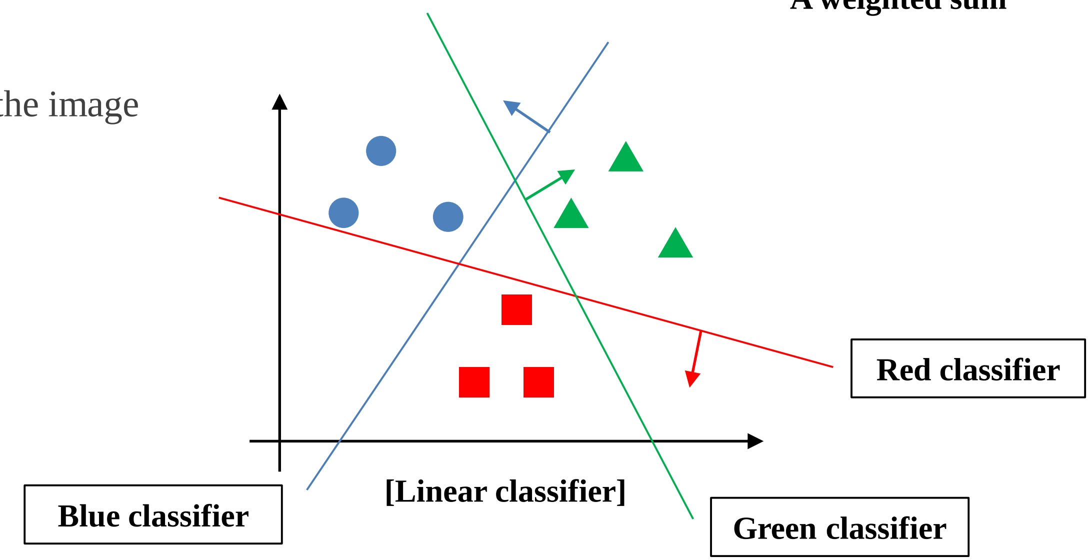
Parametric Approach: Linear Classifier

- The labels of given images is determined from the class scores that calculated by function of the pixels of given image (single column vector) and parameters (or weights), i.e., $f(x, W) = Wx + b$

[given image pixel]	[weights]	[column vector of pixel]	[bias]	[scores]
$\begin{matrix} a & b \\ c & d \end{matrix}$	$\begin{matrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{matrix}$	$\begin{matrix} a \\ b \\ c \\ d \end{matrix}$	$\begin{matrix} \alpha \\ \beta \\ \gamma \end{matrix}$	$\boxed{\begin{matrix} ax_{11} + bx_{12} + cx_{13} + dx_{14} \\ ax_{21} + bx_{22} + cx_{23} + dx_{24} \\ ax_{31} + bx_{32} + cx_{33} + dx_{34} \end{matrix}} \begin{matrix} + \alpha \\ + \beta \\ + \gamma \end{matrix}$

A weighted sum

- The linear classifier divide the image



[Linear classifier]

- **Loss Function and Optimization**
  - **Loss Function: judge the performance of classifier**
    - Given N dataset, $\{(x_i, y_i)\}_{i=1}^{N}$, where x_i is image data and y_i is integer label.

→ Loss, $L = \frac{1}{N}\sum_i L_i(f(x_i, W), y_i) = \frac{1}{N}\sum_i L_i(Wx_i, y_i)$

- Multiclass SVM Loss (Hinge Loss)

Score of wrong label

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases} = \sum_{j \neq y_i} \max(0, \boxed{s_j} - \boxed{s_{y_i}} + \boxed{1})$$

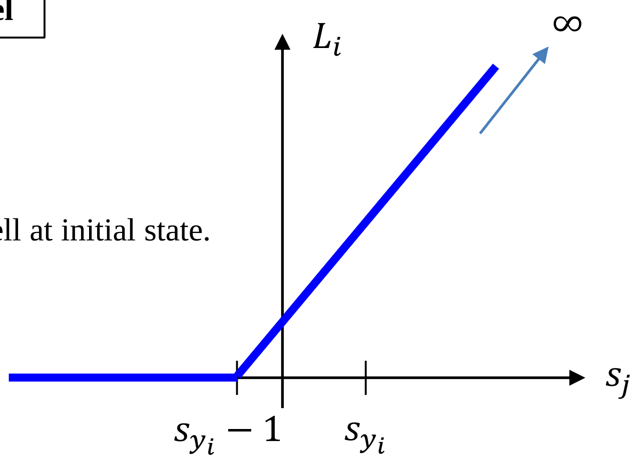
Safety margin

Score of correct label

Here, $s = f(x_i, W) = Wx_i$ is the scores vector.

- (score of correct label, s_{y_i}) $-1 <$ (score of wrong label, s_j) → $L_i > 0$
- (score of correct label, s_{y_i}) $>$ (score of wrong label, s_j) $+1$ → $L_i = 0$
- Sanity check is utilized to examine whether the learning process is going well at initial state.

→ At initialization W is small and all $s \approx 0$, thus $L = N_{\text{class}} - 1$



Loss Function and Optimization

Loss Function: judge the performance of classifier

- Regularization, $R(W)$ to prevent overfitting

$$L = \boxed{\frac{1}{N}\sum_i L_i(f(x_i, W), y_i)} + \lambda \boxed{R(W)}$$

Regularization for working well on test data

Data Loss fitted by training data

where λ is regularization strength, which is hyperparameter.

- L1 Regularization: $R(W) = \sum_{k,l} |W_{k,l}|$, sparse weights
- L2 Regularization (Weigh Decay): $R(W) = \sum_{k,l} W_{k,l}^2$, spread out weights

e.g.,

if $(1, 0)$, $R_{\mathrm{L1}} = 1$ and $R_{\mathrm{L2}} = 1$, thus both regularization have same value

if $(0.5, 0.5)$, $R_{\mathrm{L1}} = 1$ and $R_{\mathrm{L2}} = 0.5$, thus, R_{L2} has a smaller value when the weights spread out. → smaller regularization.

- Elastic net (L1 + L2): $R(W) = \sum_{k,l} \beta W_{k,l}^2 + |W_{k,l}|$
- Other regularization method: Dropout (randomly set some neurons to zero), Batch normalization

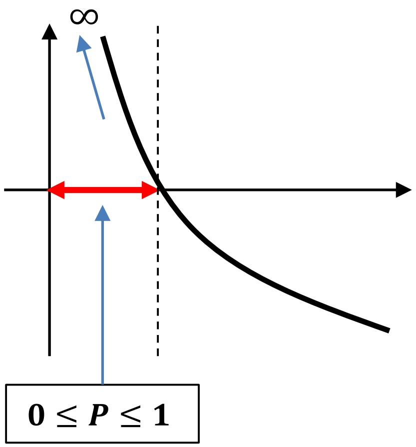
Loss Function and Optimization

Loss Function: judge the performance of classifier

- Softmax-Cross Entropy Loss (Multinomial Logistic Regression)
  - The scores is expressed by "unnormalized log probabilities" of the classes, $s = f(x_i, W)$.

$$L_i = -\log P(Y = y_i | X = x_i) = -\log \left(\boxed{\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}} \right)$$

Softmax function



  - Unnormalized log probabilities, s_j → $\exp(s_j)$ → normalize, $\exp(s_j) / \sum_j \exp(s_j)$; probabilities
  - $P = 1$, correct class → $L_i = 0$
  - $P = 0$, wrong class → $L_i \to \infty$
  - Sanity check

  ➔ At initialization W is small and all $s \approx 0$, thus $L = -\log(1/N_{\text{class}})$
- Softmax vs SVM
  - SVM has robustness by safety margin (Loss is unchangeable).
  - Softmax try to increase the probability of correct class (Loss is changeable).

Optimization: find the Weights that loss is minimized

- Gradient Descent
  - α is learning rate, which is hyperparameter.

$$\boxed{W = \alpha \cdot \frac{\partial L}{\partial W}}$$

- Stochastic Gradient Descent (SGD)
  - Minibatch gradient descent: approximate sum using a minibatch (part of the training set), not full sum, which is expensive.

Backpropagation and Neural Networks

Backpropagation

- Compute the gradient of the loss function with respect to the inputs using the Local gradient (Jacobian matrix) memorized in forward pass.
- Gate
  - Add (+) gate: gradient distributor
  - Max gate: gradient router distributing only gradient of max value
  - Mul gate: gradient switcher
  - At branches, each gradients is added.

Neural Networks (with several classifier)

- One hidden layer is responsible for one feature.
- Activation functions (non-linearities)
  - Sigmoid, $\sigma(x) = 1/(1 + e^{-x})$
  - ReLU, $\sigma(x) = \max(0,\ x)$
  - Leaky ReLU, $\sigma(x) = \max(0.01x,\ x)$
  - Maxout, $\sigma(x) = \max\left(w_1^{\mathrm{T}}x + b_1,\ w_2^{\mathrm{T}}x + b_2\right)$
  - ELU, $\sigma(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$
- "Fully-connected" Layers, FC Layer (connect to the entire input nodes.)
- The more Layers, the better capacity.
  - To prevent overfitting, regularization strength have to adjust, do not make small number of layers.

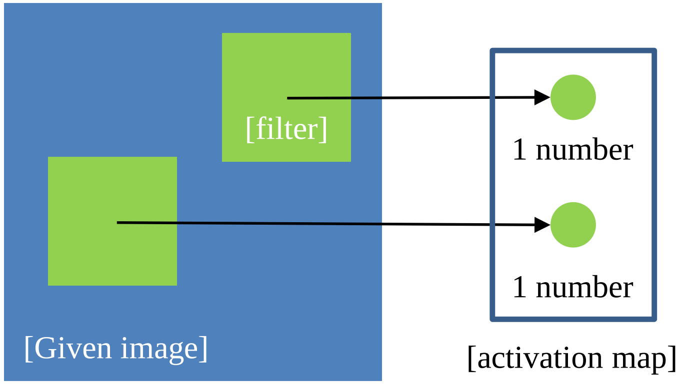
Convolutional Neural Networks (CNN)

Convolutional Neural Networks

- Consist of the Convolution Layer, Pooling Layer and FC Layer.
- Do not need to stretch into column vector $(N \times 1)$, it preserve spatial structure $(H \times W \times D)$
- *Convolve* the filter with the image, i.e., *slide the image spatially*, computing dot products $\left(w^{\mathrm{T}} x\right)$.
  - One filter (share the same weights) make one activation map
  - Output size: $(N - F)/\mathrm{stride} + 1$, where N is input size and F is filter size.
  - Using only the filter, the output size is smaller than input size, thus zero-padding, $(F - 1)/2$ is applied to preserve the output size.
  - With filter, the shrinking too fast is not good, doesn't work well. → for down-sampling, pooling layer!!
  - Convolution layer with 1×1 filter (to reduce computation cost by lower depth and for mapping)

Pooling layer for down-sampling

- no weights and no padding
- Preserve the depth of input matrix (only 1 filter)
- Max pooling
  - It transfer the max value in the range to the activation map.



Training Neural Networks

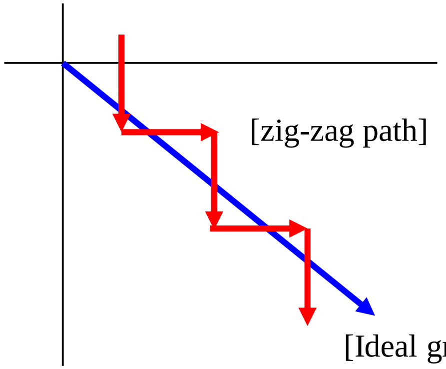
Activation functions

- Sigmoid, $\sigma(x) = 1/(1+e^{-x})$
  - Vanishing gradient make backpropagation be impossible.
  - Range [0,1] thus, not zero-centered (input, x is always positive) → slow convergence (∵ zig-zag path)

$$f\left(\sum_i w_i x_i + b\right) \rightarrow \frac{\partial f}{\partial w_i} = x_i \geq 0$$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial w_i} = \frac{\partial L}{\partial f} x_i \quad \Rightarrow \text{sign}\left(\frac{\partial L}{\partial w_i}\right) = \text{sign}\left(\frac{\partial L}{\partial f}\right)$$

$$\therefore \frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \dots, \frac{\partial L}{\partial w_i} \geq 0 \text{ or } \leq 0$$



→ The gradients on w_i are always all positive or all negative.

- tanh, $\sigma(x) = \tanh(x)$
  - Range [-1, 1] thus, zero-centered
  - Vanishing gradient occurs
- Rectified Linear Unit (ReLU), $\sigma(x) = \max(0,\ x)$
  - In positive region, vanishing gradient X (∵ $\partial\sigma/\partial x = 1$), but in negative region, vanishing gradient O.
  - Faster converge than sigmoid and tanh
  - Not zero-centered (can be solved by batch-normalizations)

Training Neural Networks

Activation functions

- Leaky ReLU,$\sigma(x) = \max(0.01x,\ x)$
  - vanishing gradient X ($\because\ \partial\sigma/\partial x = 1, x > 0\ \ \partial\sigma/\partial x = 0.01, x < 0$)
  - Faster converge than sigmoid and tanh
- Parametric Rectifier (PReLU), $\sigma(x) = \max(\alpha x,\ x)$
  - α is learned through the backpropagation.
- Exponential Linear Units (ELU) $\sigma(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$
  - *Closer* to zero-mean outputs
- Maxout, $\sigma(x) = \max\left(w_1^{\mathrm{T}}x + b_1,\ w_2^{\mathrm{T}}x + b_2\right)$
  - Vanishing gradient X
  - Doubles the number of parameters

Data Preprocessing

- Original data → zero-centered data (to alleviate the slow convergence) → normalized data, divide standard
- Zero-centered data is only used for images
  - Subtract the mean image, $[H \times W \times D]$ array (e.g. AlexNet)
  - Subtract per-channel mean, 3 numbers (e.g. VGGNet)
- Principle component analysis (PCA): find space which maximize variance of projected data (dimension ↓).
- Whitening: remove the overlap

Training Neural Networks

Weight Initialization

- *When initial weights set zero, every neuron is having same operation, thus outputs are same things and all gradient is same.*
- Small random number
  - Gaussian with zero mean and 0.01 standard deviation, $w = 0.01 * \mathrm{np.random.randn(D,\ H)}$

  → Well work for small networks, not with deeper networks where all activation become zero, thus vanishing gradient occurs.
  - 1.0 instead of 0.01

  → The outputs become either -1 or 1, thus vanishing gradient occur too.

∴ initialization too small: activation become zero → vanishing gradients
Initialization too big: activation saturate → vanishing gradients

- Xavier initialization
  - Divide the number of input, $w = \mathrm{np.random.randn(in,\ out)\ /\ np.sqrt(in)}$
  - The more number of input (N), the smaller initial weights (w).
  - It is good performance when using tanh activation function, but breaks when using ReLU.
- He et al. [2015]
  - Divide the half of input number, $w = \mathrm{np.random.randn(in,\ out)\ /\ np.sqrt(in/2)}$
  - Well work when using ReLU.

Training Neural Networks

Batch Normalization

- Make layer output in unit gaussian range before entering into following layer as input
  - Remove the instability (e.g., vanishing gradient)
  - Differentiable function → Backpropagation is okay.

$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{average}\left[x^{(k)}\right]}{\sqrt{\text{Var}[x^{(k)}]}}$$

- Insert after FC Layer or Conv Layer and before activation function
  - FC Layer: per each dimension (feature elements), average and variance are independently computed.
  - Conv Layer: per activation map (channel), average and variance are computed.
- Squash the range according need of batch normalization

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

  - γ (scaling) and β (shift) are determined by learning
  - If $\gamma = \sqrt{\text{Var}[x^{(k)}]}$ and $\beta = \text{average}\left[x^{(k)}\right]$, normalization effect X → identity mapping
- Higher learning rates and reduced dependence on initialization
- Act as regularization and reduce the need for dropout
- At test time, *moving averages (i.e., weighted averages) during at training* time is used.

[mini-batch mean]

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

[mini-batch variance]

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2$$

[normalize]

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

[scale and shift]

$$y_i = \gamma\hat{x}_i + \beta$$

Training Neural Networks

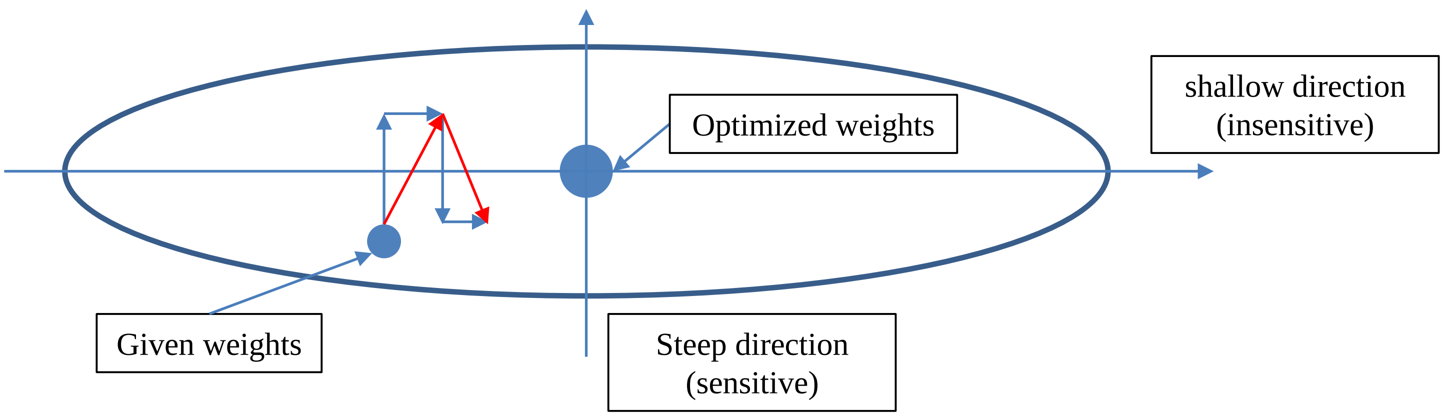
Hyperparameter Optimization

- Hyperparameter
  - Network architecture (i.e., number of hidden layer and node)
  - Learning rate
  - Regularization
- Cross-validation strategy
  - Coarse → Fine
  - Setting in log scale (e.g., $\mathrm{reg} = 10 ** \mathrm{unifrom}(-5,\ 5)$)
- Random Search vs Grid Search
  - Random Search: consider the importance of each parameter
  - Grid Search: equal interval → hard to find optimized parameters

Training Neural Networks

First–Order Optimization for parameters (i.e., weights)

- Stochastic Gradient Descent (SGD)
  - loss changes quickly in one direction (sensitive) and slowly in another (insensitive) → zig-zag path ∴ slow convergence
  - It is expressed by "*high condition number*" which is ratio of largest to smallest singular value of Hessian matrix.



  - Local minima or saddle point (more common in high dimension) → zero gradients, thus gradient descent stop.

Training Neural Networks

First–Order Optimization for parameters (i.e., weights)

- SGD + Momentum

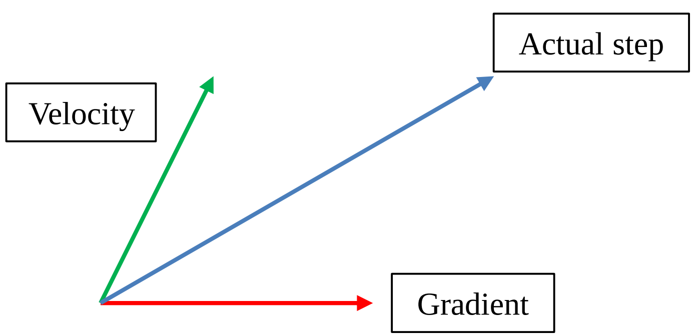
$$\boxed{\begin{aligned} v_{t+1} &= \rho v_t + \alpha \nabla f(x_t) \\ x_{t+1} &= x_t - v_{t+1} \end{aligned}}$$

where ρ is fiction coefficient (hyperparameter), which reduce the velocity, typically set as 0.9 or 0.99

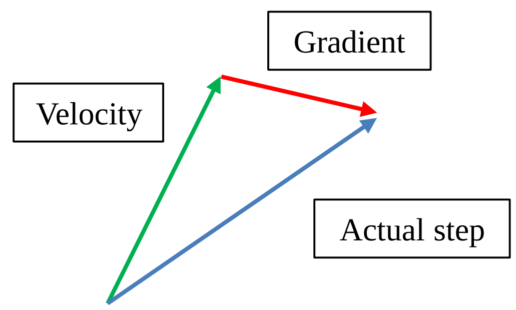
  - Steep direction → high velocity, thus sign is rapidly changeable. ∴ velocity is damped.
  - Shallow direction → the velocity is built up.

Alleviate high condition number problem

- Nesterov Momentum (Nesterov Accelerated Gradient, NAG)
  - At the spot after momentum step (velocity step), set gradient vector.



[Momentum]



[Nesterove Momentum]

Training Neural Networks

First–Order Optimization for parameters (i.e., weights)

- Nesterov Momentum (Nesterov Accelerated Gradient, NAG)

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

  - Transposition of $x_t + \rho v_t$ into $\tilde{x}_t$

$$\begin{aligned} \tilde{x}_{t+1} &= x_{t+1} + \rho v_{t+1} \\ &= x_t + (1+\rho) v_{t+1} \\ &= \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1} \end{aligned} \quad \Rightarrow \quad \begin{aligned} \therefore\ v_{t+1} &= \rho v_t + \alpha \nabla f(\tilde{x}_t) \\ \tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1} \\ &= \tilde{x}_t + v_{t+1} + \rho (v_{t+1} - v_t) \end{aligned}$$

- AdaGrad (per-parameter adoptive learning rate method)
  - Different learning rate is applied per parameters.
  - "cache" is introduced; cache is always increased. ($\because$ positive)

$$\begin{aligned} &\text{cache} \mathrel{+}= \text{gradient(x)} * \text{gradient(x)} \\ &x \mathrel{-}= \alpha * \text{gradient(x)} \,/\, (\text{sqrt(cache)} + 10^{-7}) \end{aligned}$$

  where 10^{-7} is for preventing from dividing zero.
  - Steep gradient → high cache → small learning rate (α) → update speed is lowered.
  - Gradual gradient → small cache → large learning rate (α) → update speed is increased.
  - Over long time, learning rate (α) become zero ($\because$ cache ⇑).

Training Neural Networks

First–Order Optimization for parameters (i.e., weights)

- RMSProp
  - Prevent learning rate from being zero ($\alpha \nrightarrow 0$)
  - "Decay rate" that is hyperparameter is introduced.

$$\text{cache} = \textbf{decay rate} * \text{cache} + (1 - \textbf{decay rate}) * \text{gradient(x)} * \text{gradient(x)}$$
$$x \mathrel{-}= \alpha * \text{gradient(x)} \,/\, (\text{sqrt(cache)} + 10^{-7})$$

- Adam: RMSProp with Momentum

$$\text{moment}^{\text{1st}} = \beta_1 * \text{moment}^{\text{1st}} + (1 - \beta_1) * \text{gradient(x)}$$
$$\text{moment}^{\text{2nd}} = \beta_2 * \text{moment}^{\text{2nd}} + (1 - \beta_2) * \text{gradient(x)} * \text{gradient(x)}$$
$$\text{unbias}^{\text{1st}} = \text{moment}^{\text{1st}} / (1 - \beta_1 ** \text{iter})$$
$$\text{unbias}^{\text{2nd}} = \text{moment}^{\text{2nd}} / (1 - \beta_2 ** \text{iter})$$
$$-= \alpha * \text{unbias}^{\text{1st}} \,/\, \left(\text{sqrt}\left(\text{unbias}^{\text{2nd}}\right) + 10^{-7}\right)$$

where β_1 and β_2 are hyperparameters and "unbias" is for that first and second moment start at zero.

- Learning rate (α)
  - Learning rate decay over time
  - Step decay: by half every few epochs
  - Exponential decay: $\alpha = \alpha_0 \exp(-kt)$
  - 1/t decay: $\alpha = \alpha_0/(1 + kt)$

Training Neural Networks

Second–Order Optimization

- *Hessian* as well as gradient are employed.
- No hyperparameters (e.g., learning rate α)
- Not proper to Deep Neural Network (∵ high computational cost by heavy Hessian matrix)
- BGFS (Quasi-Newton method)
  - Instead of inverting the full Hessian, approximate inverse Hessian with rank 1; Low-rank approximations
- L-BFGS (Limited memory BFGS)
  - Not form and store the full inverse Hessian
  - *It is employed after disabling all sources of noise*
  - Works very well in full batch with low stochasticity, but not in mini-batch.

Model Ensembles

- After training several models independently, the results is obtained by averaging their results at test time.
- Multiple snapshots (each result in the single model) can be averaged.
- Polyak averaging: At test time, “the exponentially decaying average of the parameter” obtained at training time (i.e., moving averages) is used. (ensembles between the parameters)

Training Neural Networks

Regularization

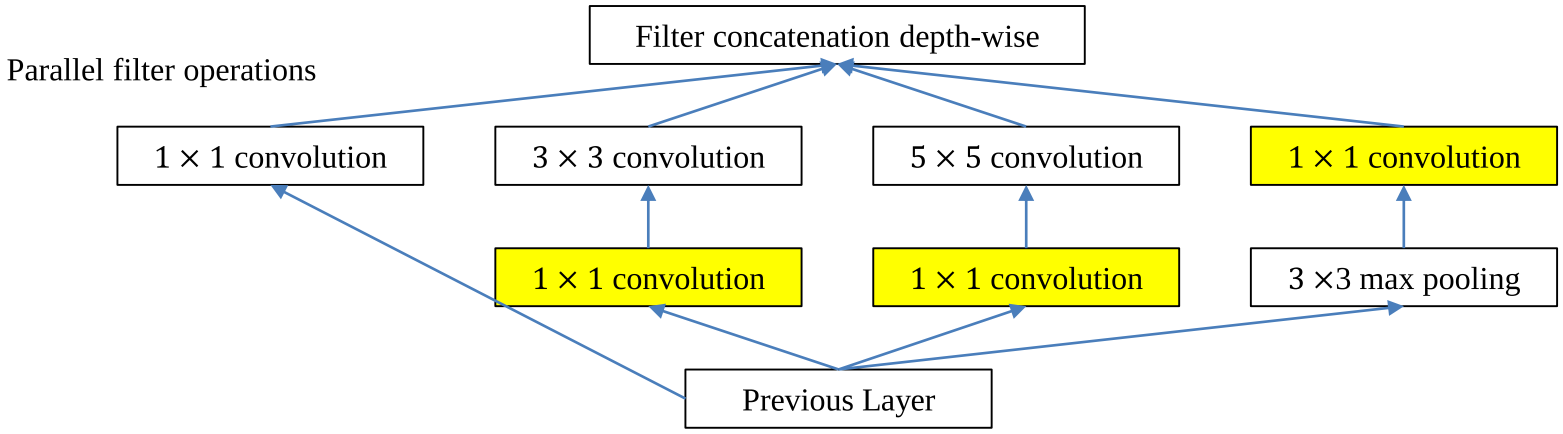
- Dropout: regarded as Ensemble
  - It is considered that each binary mask (according to dropout neuron location) is one model
  - At test time, the result have to multiply by dropout probability; compensated by scaling activations,
    all neurons are active! (no drop out)
  - *Inverted dropout: instead of multiply probability at test time, divide probability at training time.*
- Data Augmentation: deform image pixel
  - Horizontal Flips
  - Random crops and scales
  - Color Jitter) 1. PCA to [R, G, B]
    2. sample a color offset along principal component directions
    3. offset to all pixels

- **CNN Architectures**
  - **LeNet-5 [LeCun et al., 1998]**
  - **AlexNet [Krizhevsky et al., 2012]**
    - FC7 Layer: FC Layer just before classifier
    - Details
      - Activation function: ReLU
      - Norm Layers
      - Data augmentation
  - **ZFNet**
    - Smaller filter size, larger the number of filters than AlexNet
  - **VGGNet**
    - Fixed filter (3×3 CONV stride 1 with pad 1 and 2×2 MAX POOL stride 2)
    - Smaller filter size (3×3), deeper networks (non-linearities ↑ by more activation functions)
      - same effective receptive field: three 3×3 CONV = one 7×7 CONV
      - Have fewer parameters

CNN Architectures

GoogleNet

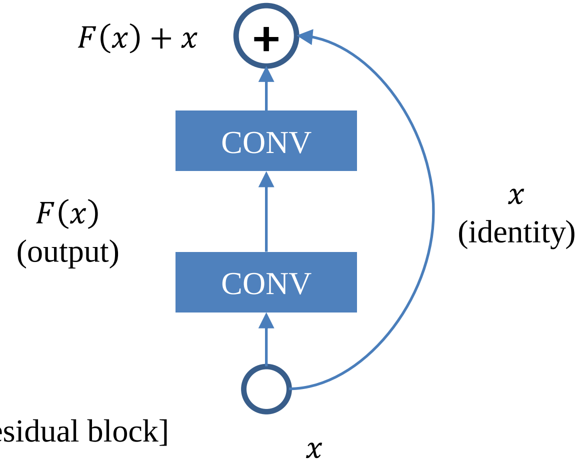
- Instead of FC layers, average pooling is used. → reduce parameters
- "Inception" module
  - Local network topology: network within a network
  - Very expensive computation cost (∵ pooling layer preserve the depth of input, thus total depth grow at every layer)
  - ➔ *Bottleneck layers (yellow)* that use 1 × 1 convolutions to preserve spatial dimensions and reduce depth is introduced. (i.e., projects depth to lower dimension, the number of filter)
- Auxiliary classification outputs give additional gradient at lower layers to prevent vanishing gradients.



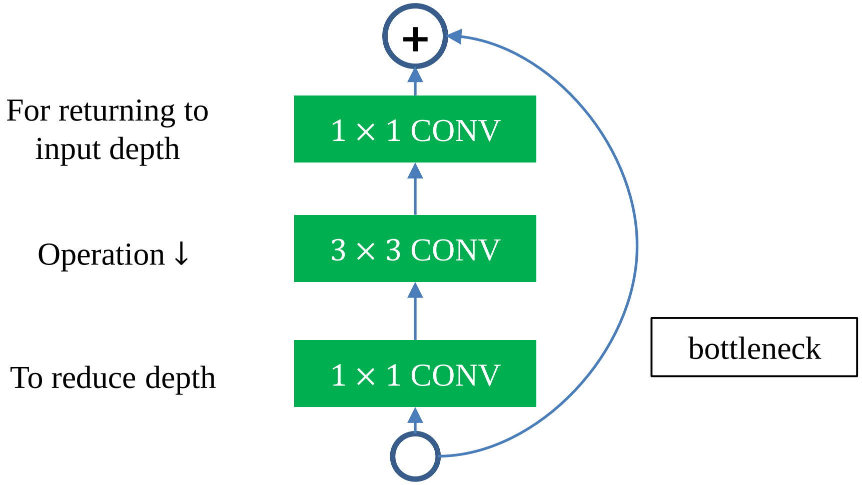
CNN Architectures

ResNet

- Residual connections: skip connection
- The more layer, the better performance unlike previous architectures (e.g., AlexNet, VGGNet)
  - Solution: copying the learned layers from the shallow model and setting the other layers to identity mapping.
  - Use layers to fit *residual*, $F(x) = H(x) - x$ (input data), which is variance about x, instead of $H(x)$ directly
  - If weights in residual block are zero, the block is identity mapping (not learned), thus other blocks are same in shallow model.
- Stack residual blocks, which has two 3×3 CONV layers each.
- Use global average pooling instead of FC layers at the end and only use FC 1000 to output classes
- For deeper networks, use bottleneck (1×1 CONV) to reduce cost
- Batch normalization (higher learning rate, drop out X) after every CONV layer



[Residual block]



CNN Architectures

Network in Network (NiN)

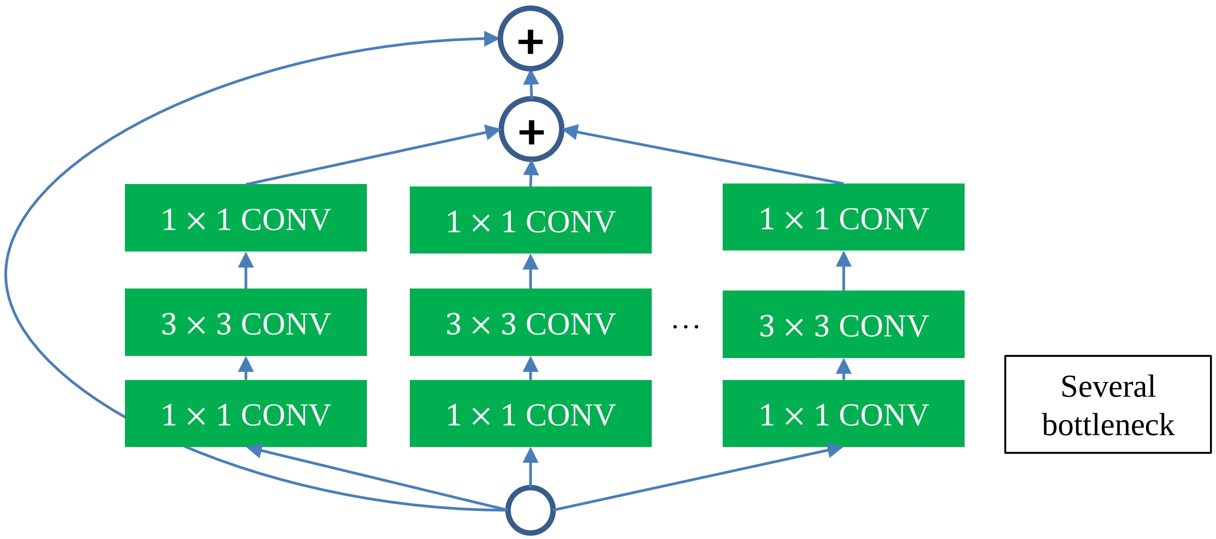
- MLP (Multi Layer Perception), which stack FC Layers, within each CONV layers.

Wide Residual Networks [Zagoruyko et al. 2016]

- Residuals are the important factor itself, not depth. → wider residual blocks (F × k filter instead of F)
- Increasing width is more computationally efficient than depth.

ResNeXt

- Increasing width of residual block by using several bottleneck (i.e., cardinality), which is similar to inception module in GoogleNet, instead of single bottleneck



CNN Architectures

Deep Networks with Stochastic Depth

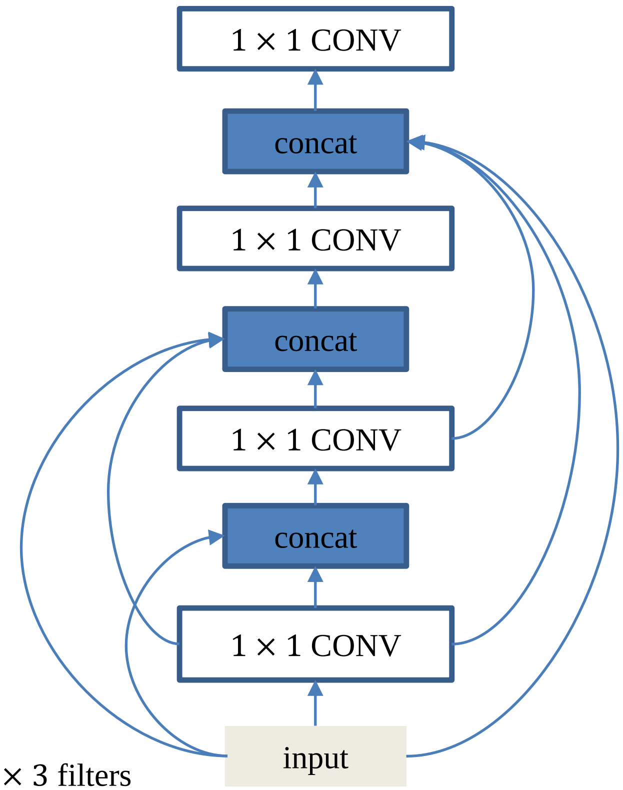
- Vanishing gradient and training time ↓
  - Randomly drop a subset of layers, i.e., residual block in ResNet (similar to dropout, which drops the nodes randomly)
  - ➔ some networks to identity connection
- Use full network without dropping any subsets at test time

FractalNet

- Residual connection X
- Trained with dropping out sub-paths, but use full network at test time

Densely Connected Convolutional Networks (DCN?)

- Dense blocks
  - Each layer is connected to every other layer in feedforward path
- Vanishing gradient ↓
- feature propagation ↑, feature reuse ↑
  (∵ output of each layer is used at other layer.)
- Input image data is used at every layer



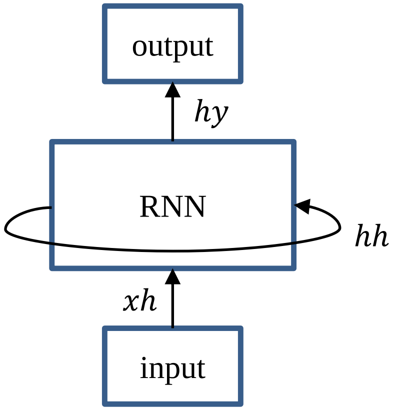
[Dense Block]

SqueezeNet

- Fire modules
  - ‘Squeeze’ layer with 1 × 1 filters feeding (→) ‘expand’ layer with 1 × 1 and 3 × 3 filters

Recurrent Neural Network

Recurrent Neural Network, RNN



- From input data at each time step, to "*Hidden state*"
- Goal: predict a output vector at some time steps

$$\boxed{h_t = f_W(h_{t-1}, x_t)}$$

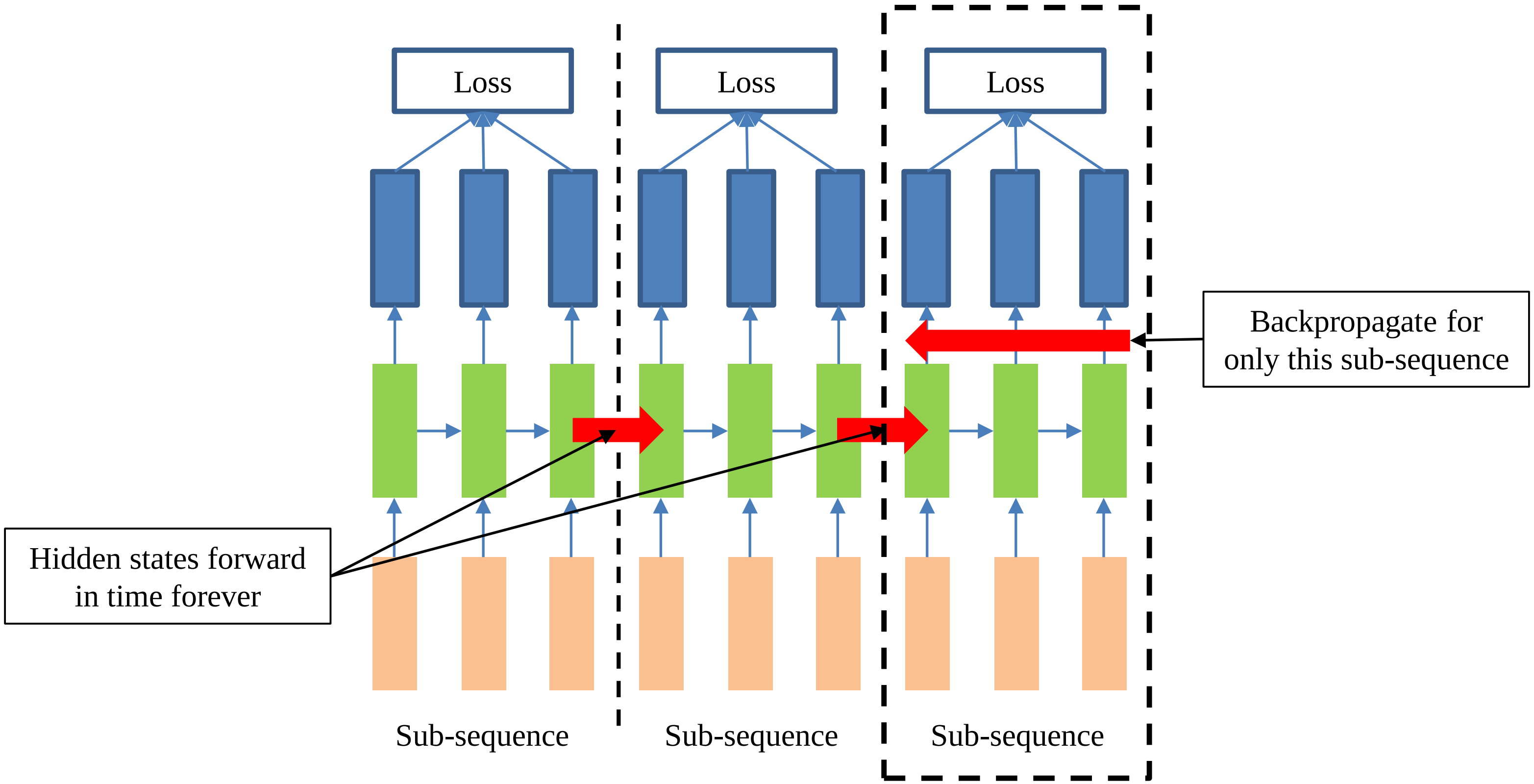
$$\boxed{h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)} \rightarrow \boxed{y_t = W_{hy}h_t}$$

  - where h_t is new state (updated hidden state), h_{t-1} is old state and x_t is input vector at some time step.
  - The function, f_W and its parameters, W are same used at every time step (i.e., Re-use the same weight matrix at every time step).
- Many to Many: size(input) = size(output)
  - Through "Ground truth" at each step, $\text{Loss}(y_t)$ can be obtained separately. → $\sum \text{Loss}_t$
- Many to One
  - The only output at the final hidden state (e.g., if step range [0, t], $\text{Loss}(y_t)$)
- Sequence to Sequence (Many to One + One to Many): size(input) ≠ size(output)
  - Many to One: "Encode" input sequence in a single vector
  - One to Many: "Decode" output sequence from single input vector

Recurrent Neural Network

Recurrent Neural Network, RNN

- Truncated Backpropagation
  - Forward and backward through sub-sequence of the sequence instead of whole sequence
  - Hidden states forward in time forever, but only backpropagate for some smaller number of steps (sub-sequence)

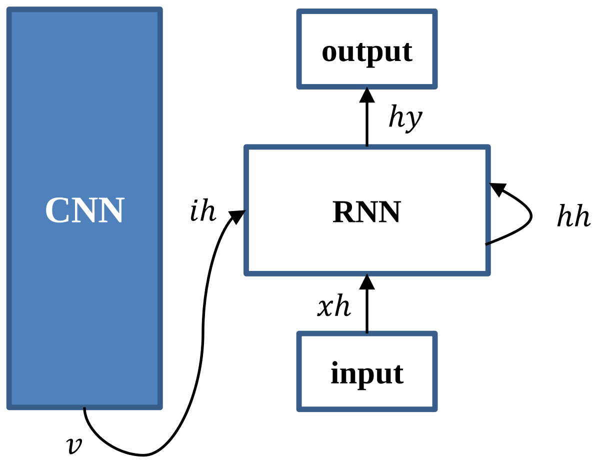


Recurrent Neural Network

Image Captioning

- CNN (for image process) + RNN (for sequence process)
  - At the final stage of CNN, FC 1000 and softmax are not used for class score, but dates just transfer to RNN

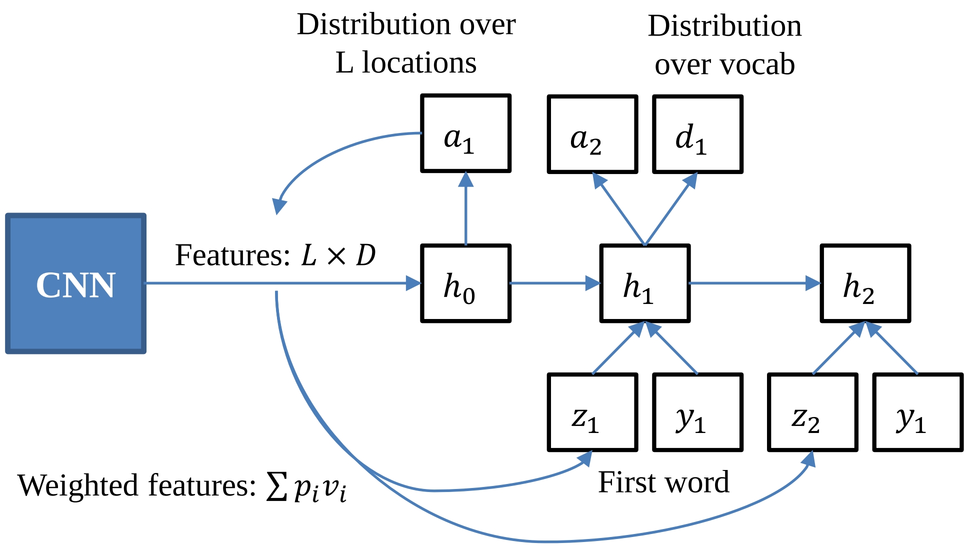
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + W_{ih}v)$$



Recurrent Neural Network

Image Captioning with Attention

- Generate each word according to a different spatial location
  - Distribution over locations + distribution over word
- Soft attention
  - Summarize all locations, $z = p_a a + p_b b + p_c c + p_d d = \sum p_i v_i$ where p_k is probability of distance and k is feature
  - Better to use gradient descent
- Hard attention
  - Only one location (that is the highest probability)
  - Gradient descent X



Recurrent Neural Network

$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$

Other RNNs

- Multilayer RNNs with several hidden states (↔ single layer RNN)
  - At same layer, the same parameters, but different weights between other layers
  - Cell state (stacked hidden state) X

Input from previous hidden state

$$h_t^l = \tanh\left(W_{hh}^l h_{t-1}^l + W_{h^*h}^l h_t^{l-1}\right) = \tanh W^l \begin{pmatrix} h_{t-1}^l \\ h_t^{l-1} \end{pmatrix}$$

where l denotes the layer number

- Long Short Term Memory (LSTM) similar to ResNet in CONV
  - Cell state (stacked hidden state) O
  - i: input gate, *whether* write input x_t (e.g., on:1 or off:0)
  - f: forget gate, how much forget cell state at previous step
  - o: output gate, how much reveal cell state, c_t
  - g: gate gate, *how much* include input cell (e.g., portion of input)
  - reduce vanishing gradients or exploding gradients because Backprop ($c_t - c_{t-1} \rightarrow \cdots c_0$) only *elementwise multiplication by f (forget gate), which is changeable every step*, no matrix (W) which is unchangeable at every step and *only one activation function* is applied.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_{t-1}^l \\ h_t^{l-1} \end{pmatrix}$$

Cell state $\quad c_t^l = f \odot c_{t-1}^l + i \odot g$

hidden state $\quad h_t^l = o \odot \tanh(c_t^l)$

cf., These occur in vanilla RNN because Backprop ($h_t \rightarrow h_{t-1} \rightarrow \cdots \rightarrow h_0$) multiplies by weights (W_{hh}^{T}) every RNN cells: Vanishing, max(singular value) < 1 or Exploding, max(singular value) > 1 and activation function is applied every step

- Exploding gradients prevented by Gradient clipping

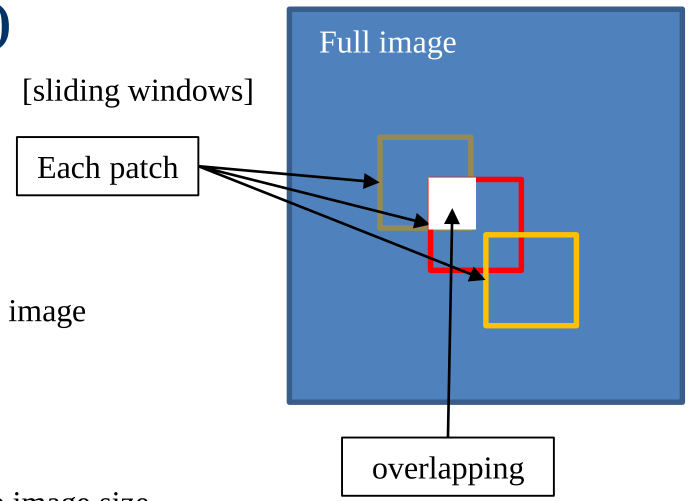
[Gradient clipping]

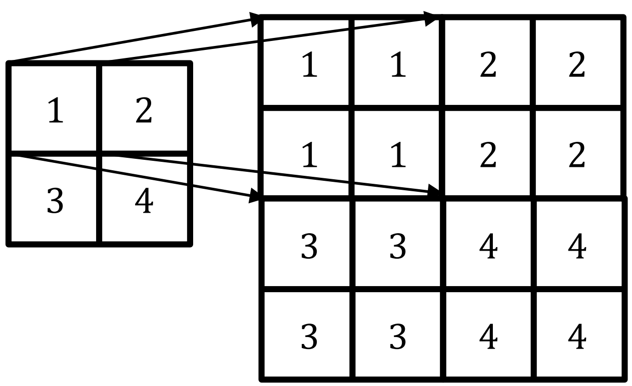
$$\text{norm} = \sum(\text{grad} * \text{grad})$$

$$\text{if norm} > \text{threshold: grad} \mathrel{*}= (\text{threshold / norm})$$

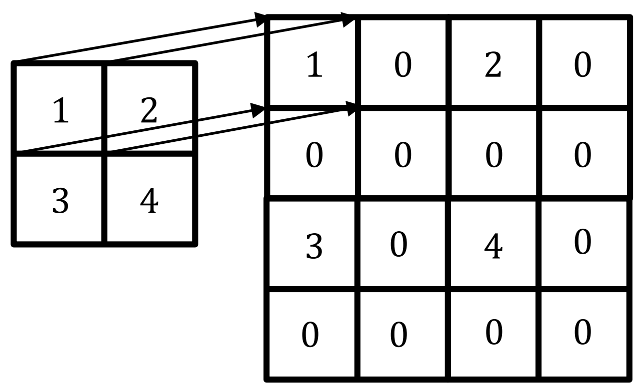
- **Segmentation; pixel (↔ classification; image)**
  - **Semantic –**
    - No objects, just pixels
    - don't differentiate instances
    - Sliding windows
      - Classify center pixel on each small region (patch) extracted by input full image
      - Inefficient and no reusing shared features between overlapping patches
    - Fully Convolutional (FCN)
      - Down-sampling with max polling and strides is applied for lower cost.
      - Up-sampling (e.g., Nearest Neighbor, Bed of Nails) is used to restore the image size.

[sliding windows]





[Nearest Neighbor]



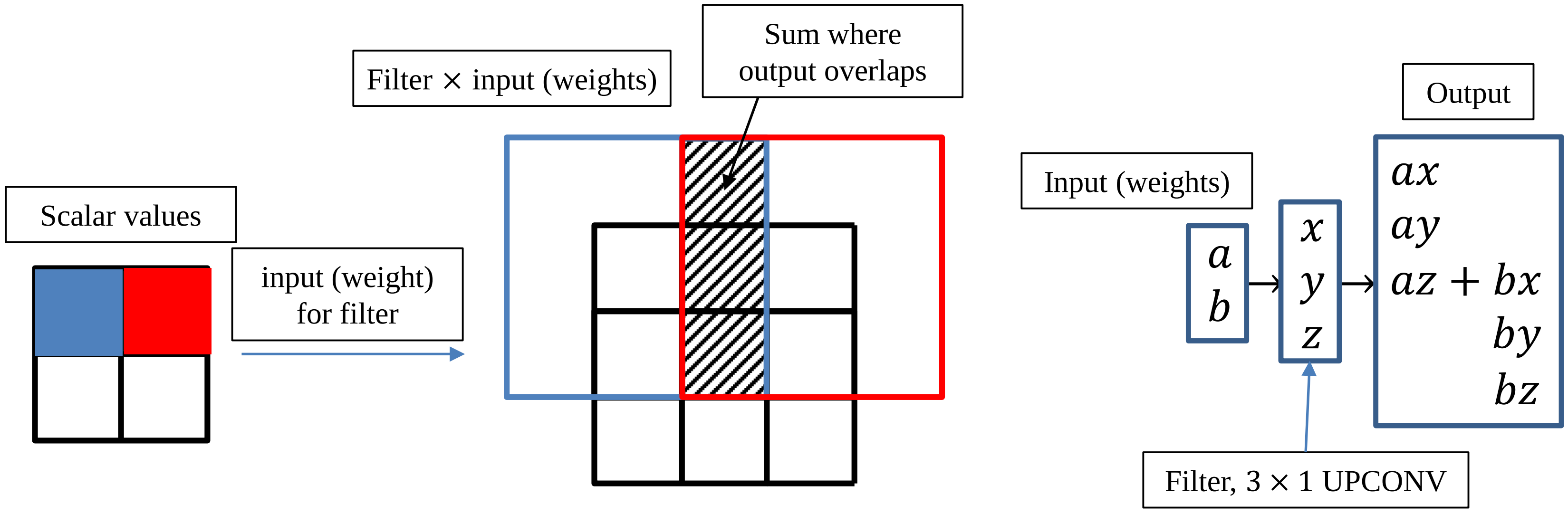
[Bed of Nails]

Segmentation; pixel (⟷ classification; image)

Semantic –

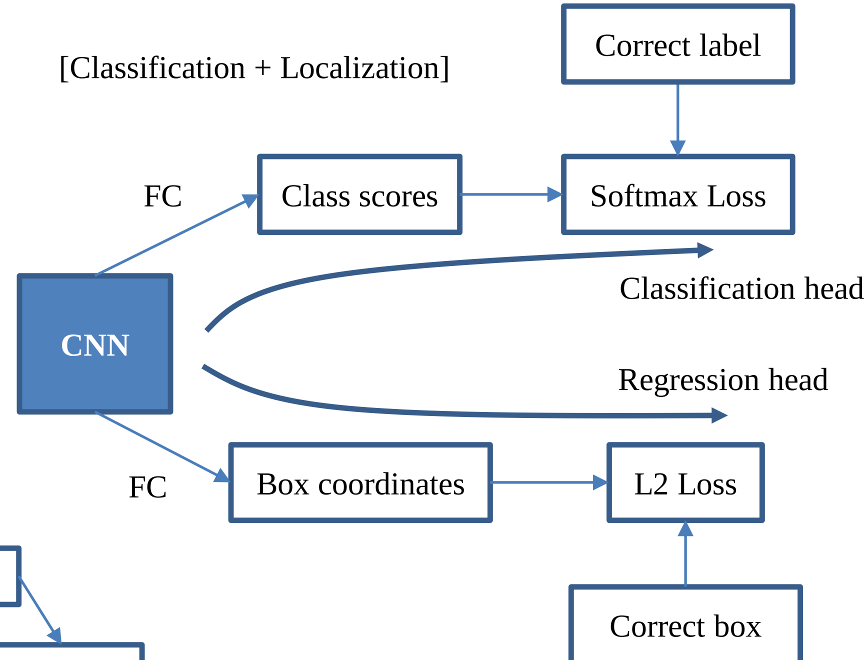
- Fully Convolutional (FCN)
  - Max unpooling that remember which element was max during down-sampling and use this position for up-sampling; other positions fill zero (e.g., [(1,2,3,5)] → [5] → … → [output(5)] → [(0,0,0,output(5))]).
    (∵ corresponding pairs of down-sampling and up-sampling layers)
  - Transpose convolution (Upconvolution, Fracctionally strided convolution and Backward strided convolution)

cf., backward in CONV = forward in UPCONV, forward in CONV = backward in UPCONV

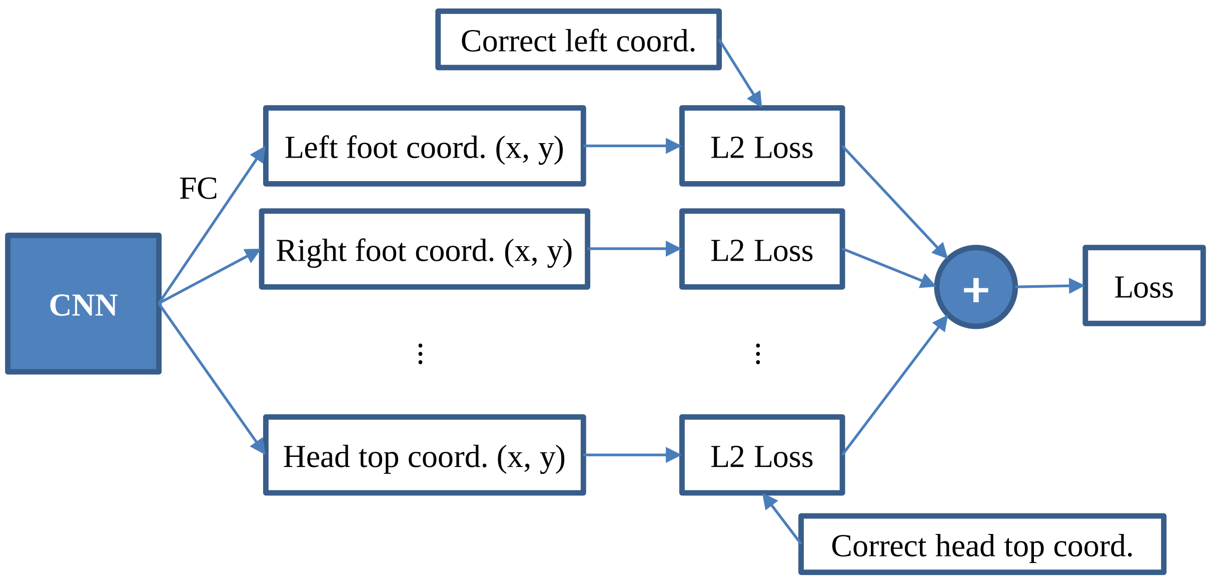


[Transpose Convolution]

- **Multitask Loss**
  - **Classification + Localization**
    - Classification
      - input image → class label
      - Measure: accuracy
    - Localization; regression
      - input image → box coordinates (x, y, w, h)
      - Measure: IOU (Intersection Over Union)
  - **Human Pose Estimation**



[Classification + Localization]



[Homan Pose Estimation]

Object Detection; multiple object

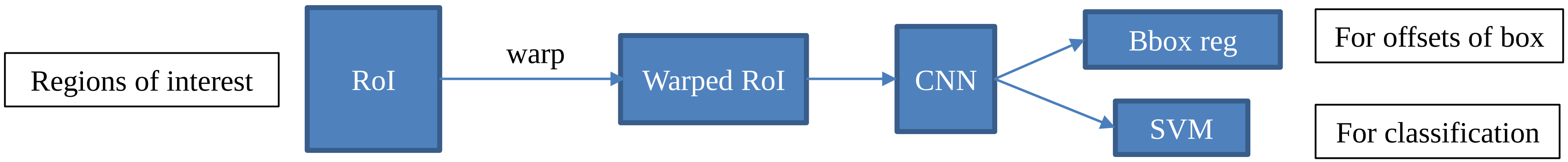
As Regression

- Many number of outputs
  - Dependent on the number of object (N) → output number = $N \times 4$
  - e.g., 1 object → (x, y, w, h) / 2 objects → (x_1, y_1, w_1, h_1) (x_2, y_2, w_2, h_2)
- Generally unknown number of object in advance

As Classification

- Sliding windows
  - Apply CNN to each patch of the image and classifies each patch as object or background
  - Hugh number of patch (e.g., locations and scales) → cost ↑ (∵ CNN operations ↑)

  → *Region Proposals, which find blobby image regions that are likely to contain objects* is introduced
- Region-based CNN (R-CNN)
  - 2K Region proposals (i.e., regions of interest, RoI) are given from a selective search from input whole image.
  - Extracted regions have different size → image regions are *warped to same size* for feeding CNN input
  - Classify the regions with SVM (hinge loss)
  - Linear Regression for bounding box offsets (revision of region proposals)

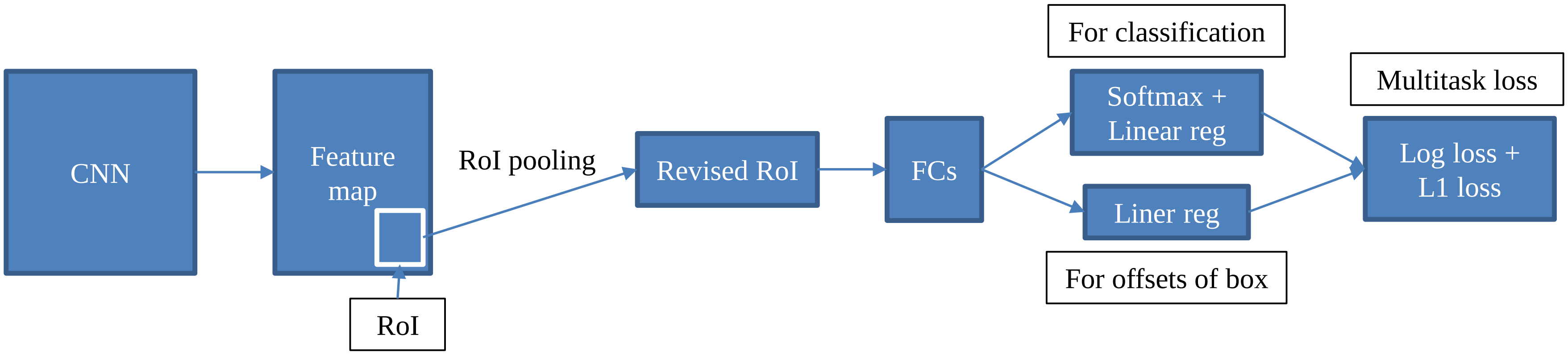


Object Detection; multiple object

As Classification

Fast R-CNN

- Forward whole image through CNN (↔ each region at R-CNN)
- RoIs are extracted from feature map of image.
- RoI pooling, which is similar to the max pooling to revise the image size for input of FC layer
- After FC layer, the softmax and regression are applied.
- Runtime dominated by region proposals: “bottle neck”



Object Detection; multiple object

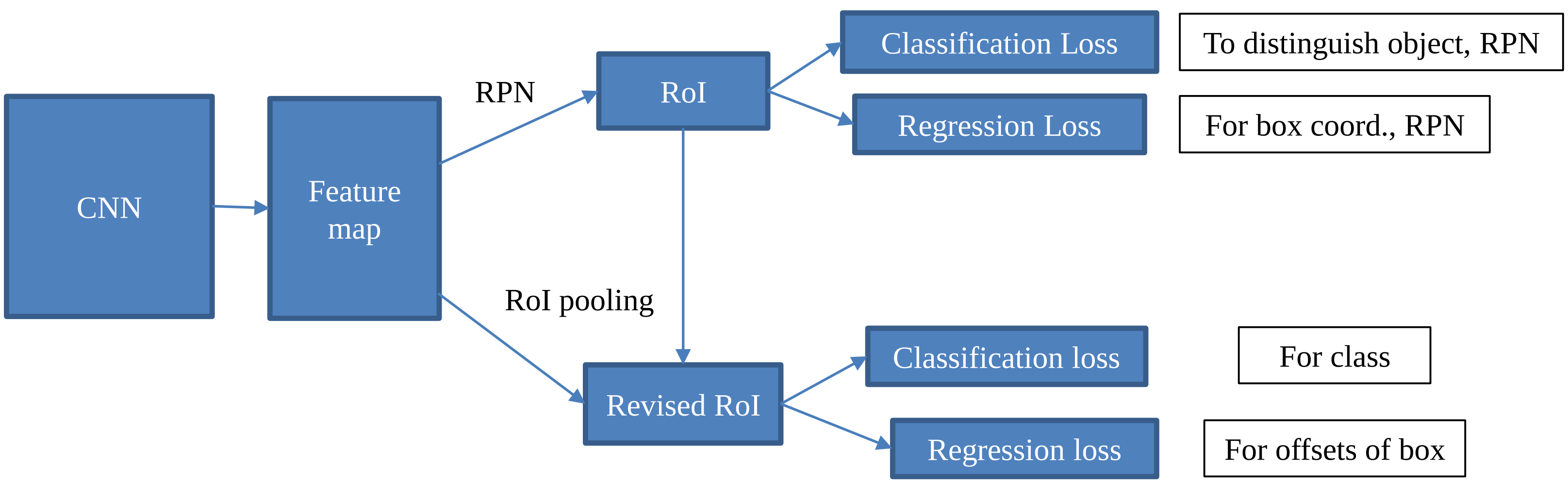
As Classification

Faster R-CNN

- Solve the problem of bottleneck in Fast R-CNN
- Add “Region Proposal Network, RPN” to predict RoI from features
  - RPN classify object / not object
  - RPN regress box coordinates
  - Final classification score (object classes)
  - Final box coordinates (offsets of box)

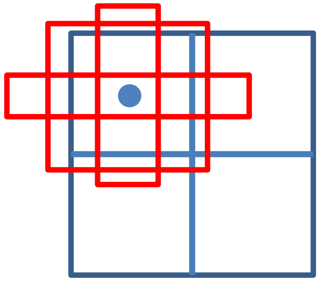
Added RPN

Same as Fast R-CNN



Object Detection; multiple object

As Classification



- YOLO (You Only Look Once), SSD (Single Shot Detection)
  - Without RoI, only one big CNN is applied.
  - Input image is divided into grid → making a set of base boxes centered at each grid cell.
  - Regress from each of the base boxes (red box) to a final box with [dx, dy, dh, dw, confidence (i.e., possibility to include the object and accuracy of predicted thing by the box)]

  cf., if there is not any object → confidence = 0, otherwise (i.e., exist objects), confidence = IoU between predicted box and ground truth

  - Predict classification scores for each of classes (+ background as a class)

Segmentation; pixel (↔ classification; image)

Instance – (semantic + detection)

- Differentiate instances → pixel labeling in each instance
- Mask R-CNN
  - Similar to Faster R-CNN (CNN-RPN-…)
  - Semantic segmentation for each RoI