Bulletin Points on Fundamental Deep Learning

Julique

December 8, 2023

1. Faster optimizer.

   Originally, the gradient descent is

   $$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}).$$

   - Momentum optimization: cares a great deal about previous gradients.

     (a) $\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$;

     (b) $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{m}$.

     Overall, $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ where $\boldsymbol{m}$ is called *momentum vector*, β is called *momentum*. If the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate η multiplied by $\frac{1}{1-\beta}$ (ignoring the sign).

   - Nesterorv accelerated gradient (Figure 1): measures the gradient of the cost function not at the local position $\boldsymbol{\theta}$ but slightly ahead in the direction of the momentum, at $\boldsymbol{\theta} + \beta \boldsymbol{m}$.

     (a) $\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta \boldsymbol{m})$;

     (b) $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{m}$.

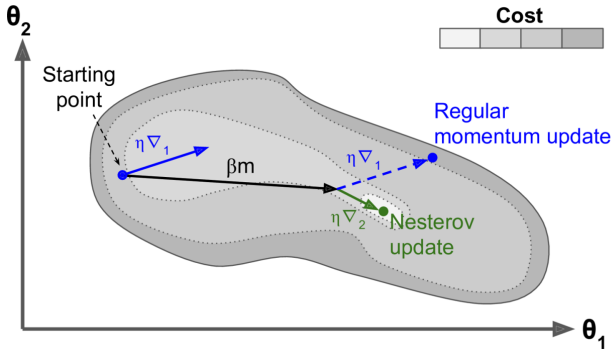

Figure 1: Nesterorv accelerated gradient.

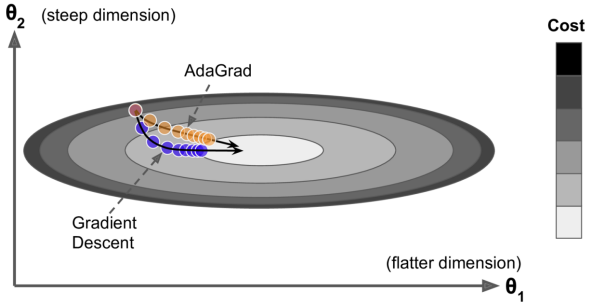

Figure 2: AdaGrad

- AdaGrad (Duchi et al., 2011): corrects the direction earlier to a point a bit more toward the optimum. AdaGrad performs well for simple quadratic problems, but it often stops too early when training neural networks (Figure **??**).
  (a) $\boldsymbol{s} \leftarrow \boldsymbol{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$;
  (b) $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \varepsilon}$.[1]
  In other words, $\theta_i \leftarrow \theta_i - \frac{\eta}{\sqrt{s_i + \left[\nabla_{\theta_i} J(\boldsymbol{\theta})\right]^2 + \varepsilon}} \nabla_{\theta_i} J(\boldsymbol{\theta})$.
- RMSProp: prevents slowing down too fast in AdaGrad by accumulating only the gradients from the most recent iterations by adding a decay rate β.
  (a) $\boldsymbol{s} \leftarrow \beta \boldsymbol{s} + (1-\beta) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$;
  (b) $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \varepsilon}$.
- Adam (Kingma & Ba, 2014): combines the ideas of momentum optimization and RMSProp.
  (a) $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1-\beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$;
  (b) $\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1-\beta_2) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$;
  (c) $\hat{\boldsymbol{m}} \leftarrow \frac{\hat{\boldsymbol{m}}}{1-\beta_1^t}$;
  (d) $\hat{\boldsymbol{s}} \leftarrow \frac{\hat{\boldsymbol{s}}}{1-\beta_2^t}$;
  (e) $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \hat{\boldsymbol{m}} \oslash \sqrt{\hat{\boldsymbol{s}} + \varepsilon}$.
- AdaMax: uses ℓ_∞ norm instead of Adam's ℓ_2 norm to scale down the parameters.
  (a) $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1-\beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$;
  (b) $\boldsymbol{s} \leftarrow \max(\beta_2 \boldsymbol{s}, |\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})|)$;
  (c) $\hat{\boldsymbol{m}} \leftarrow \frac{\hat{\boldsymbol{m}}}{1-\beta_1^t}$;
  (d) $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \hat{\boldsymbol{m}} \oslash \boldsymbol{s}$.
- Nadam optimizer: is Adam optimization plus the Nesterov trick, i.e.,

$$\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1-\beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta_1 \boldsymbol{m}).$$

2. Learning rate scheduling.
   - Power scheduling: $\eta(t) = \eta_0 \left(1 + \frac{t}{s}\right)^{-c}$.
   - Exponential scheduling: $\eta(t) = \eta_0 \left(\frac{1}{10}\right)^{\frac{t}{s}}$.
   - Piecewise constant scheduling: a step function.
   - Performance scheduling: reduces the learning rate by a factor when the error stops dropping.
   - 1-cycle scheduling (Smith, 2018): during the whole training period, learning rate ↑ then ↓, while the momentum ↓ then ↑.
3. Regularization.
   - ℓ_1 and ℓ_2 regularization: Adding a penalty term on weights, i.e., $\lambda \|w\|_1, \lambda \|w\|_2$ or their linear combination (elastic net).
   - "Dropout": Randomly letting some neurons die (normalizing the output by the *keep probability* is necessary).
   - MC (Monte Carlo) Dropout (Gal & Ghahramani, 2016): Averaging over multiple predictions with dropout gives us a Monte Carlo estimate that is generally more reliable than the result of a single prediction with dropout off.
   - Max-norm regularization: putting a constraint that $\|w\|_2 \leqslant r$ or rescaling by $w \leftarrow w \frac{r}{\|w\|_2}$.

[1] The operator $\odot$ and $\oslash$ mean element-wise multiplication and element-wise division.

- Data augmentation.

4. Why do we need reconstruction? To preserve the core information of inputs.
   - Feature Extraction: Use an autoencoder to reduce the dimensionality of image data. For example, encode high-resolution images into smaller feature vectors for image search or classification
   - Denoising: Train an autoencoder to remove noise from mobile phone photography images. Users upload damaged images, and the autoencoder outputs a clearer version.
   - Regularization: When training a classification network on a small image dataset, use the reconstruction loss of an autoencoder as a regularization term to prevent overfitting.
5. One-hot encoding is for a **small** number of categories. For **large** vocabulary, it is more efficient to encode by embeddings.
6. Key elements of convolutional neural networks (CNNs).
   - Convolutional layers: one filter for one feature (one channel, or one matrix), and much fewer parameters than the dense layers.
   - Pooling layers: subsample or reduce the dimensionality.
   - Architecture: Convolutional layer(s) $\rightarrow$ Pooling layer(s)$\rightarrow$ Convolutional layer(s) $\rightarrow$ Pooling layer(s) $\rightarrow \cdots \rightarrow$ Convolutional layer(s) $\rightarrow$ Pooling layer(s)$\rightarrow$ Fully connected layers.
7. Classic examples on CNNs.
   - LeNet-5: CNNs with average pooling.
   - AlexNet (Krizhevsky et al., 2012): CNNs (with max pooling) + dropout + data augmentation (randomly shift the training images) + local response normalization (LRN)[2].
   - ZF Net: AlexNet with fewer hyper-parameters.
   - GoogLeNet (Szegedy et al., 2015) (for classification).
     - GoogLeNet is much deeper than AlexNet, is with less dense layers, and thus has 10 times fewer parameters than AlexNet (roughly 6 million instead of 60 million).

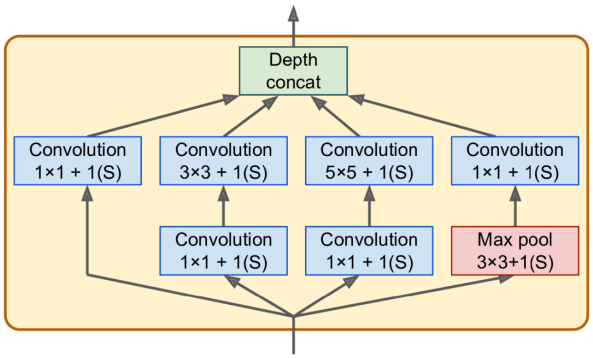

Figure 3: Inception module.

[2]The most strongly activated neurons inhibit other neurons located at the same position in neighboring feature maps (such competitive activation has been observed in biological neurons):

$$b_i = a_i \left(k + \alpha \sum_{j=\underline{j}}^{\overline{j}} a_j^2 \right)^{-\beta}$$

where $\overline{j} = \min\left(i + \frac{r}{2}, f_n - 1\right)$, $\underline{j} = \max\left(0, i - \frac{r}{2}\right)$, b_i is the normalized output of the neuron in feature map i, a_i is the activation of neuron i before normalization, f_n is the number of feature maps, and k (bias), r (depth radius), α, β are all hyper-parameters.

– Basic element (which decreases the parameters): the *inception module* [3] ≈ **a convolutional layer on steroids** that can output feature maps that capture complex patterns at various scales (Figure 3).

– Overall architecture (See Figure 4).

∗ A bottleneck layer (the left two connected blue layers): increases the feature extraction capability of NNs without adding many parameters.

∗ The global average pooling layer outputs the mean of each feature map and thus drops any remaining *spatial information* (It is fine since there was not much spatial information left at this point).[4]

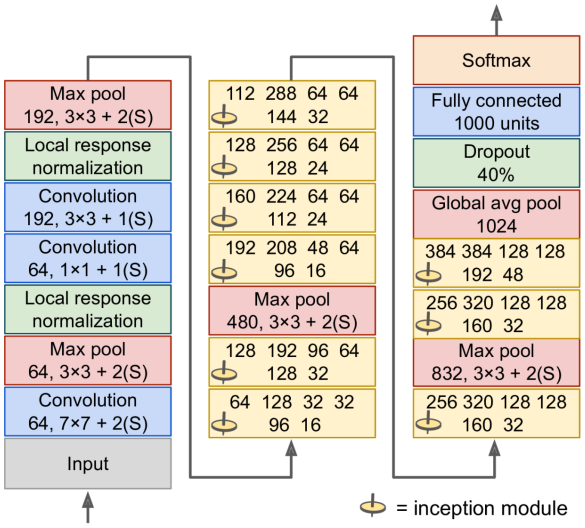

Figure 4: GoogLeNet architecture.

- ResNet (He et al., 2016): an extremely deep CNN composed of 152 layers.

  – It confirmed the general trend: **models are getting deeper and deeper, with fewer and fewer parameters** by *skip connections* or *shortcut connections* (Figure 5).

$$\underbrace{h\left(\boldsymbol{x}\right)}_{\text{Output}} = \underbrace{\boldsymbol{x}}_{\text{Input}} + \underbrace{h\left(\boldsymbol{x}\right) - \boldsymbol{x}}_{f(\boldsymbol{x}),\text{What ResNet learns}} .$$

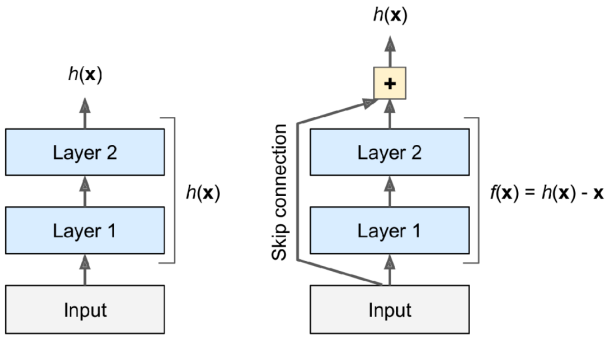

Figure 5: Residual unit (right) forces the model $f(\boldsymbol{x}) = h(\boldsymbol{x}) - \boldsymbol{x}$.

– Architecture (Figure 6):

∗ To fill the gap in the middle between the blue and purple residual units, we need the red dashed directed line as Figure 7.

[3] The notation "$3 \times 3 + 1$(S)" means that the layer uses a 3×3 kernel, stride 1, and "same" padding.

[4] The spatial information is the information of a picture as a 2D representation. For example, a flattened transformation will drop the spatial information.

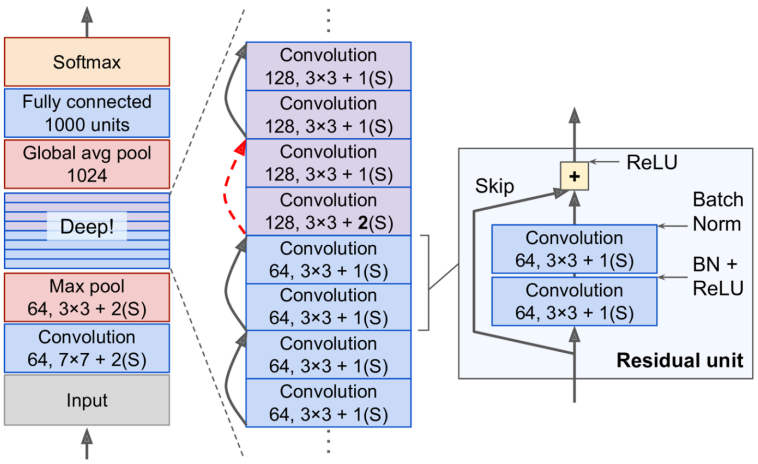

Figure 6: ResNet Architecture.

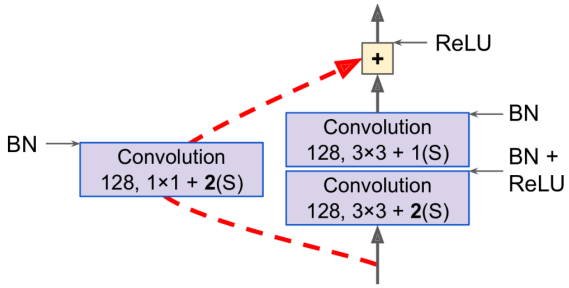

Figure 7: Skip connection when changing feature map size and depth.

  - ResNet-34 is the ResNet with 34 layers (only counting the convolutional layers and the fully connected layer)17 containing 3 residual units (RUs) that output 64 feature maps, 4 RUs with 128 maps, 6 RUs with 256 maps, and 3 RUs with 512 maps.
  - ResNets deeper than that, such as ResNet-15, have different structures on RUs.
- Inception-v4: merges the ideas of GoogLeNet and ResNet.
- Xception (*Extreme Inception*, a variant of the GoogLeNet): also merges the ideas of GoogLeNet and ResNet, but it replaces the inception modules with a special type of layer called a *depthwise separable convolution layer*.
  - Regular convolutional layers: try to simultaneously capture spatial patterns (e.g., an oval) and cross-channel patterns
  - Separable convolution layers (Figure 8): spatial and cross-channel patterns can be modeled separately.
    * The first part works for spatial patterns, applying a single spatial filter for each input feature map. [5]
    * The second part works for cross-channel patterns, which is a regular convolutional layer with 1×1 filters (similar to GoogLeNet).
- SENet (Squeeze-and-Excitation Network): extends existing architectures such as inception networks and ResNets by adding a small NN called an *SE block* (Figure 9).
- The SE blocks help recalibrate (**extract relevant ones and filter out useless ones**) the features by focusing only on the depth dimension and **adjusting weights on different features**.

[5]Note: Avoid using spatial-only filters on very few channels (Figure 8 is just for illustration). Therefore, Xception starts with 2 regular convolutional layers to generate many channels.

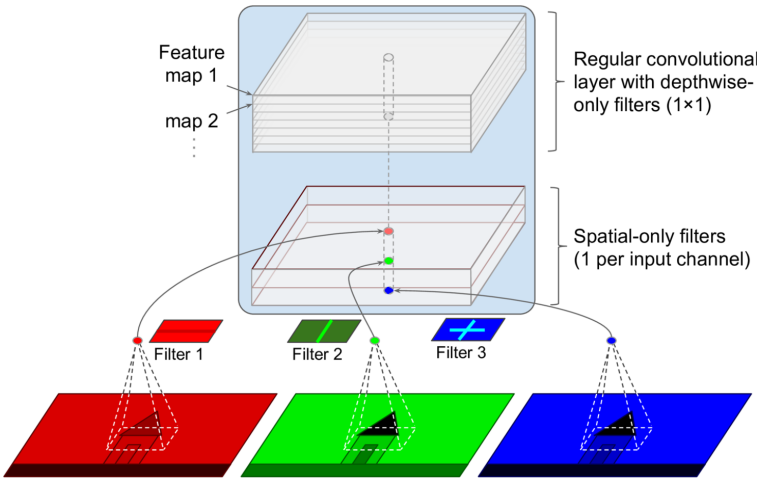

Figure 8: Separable convolutional layer.

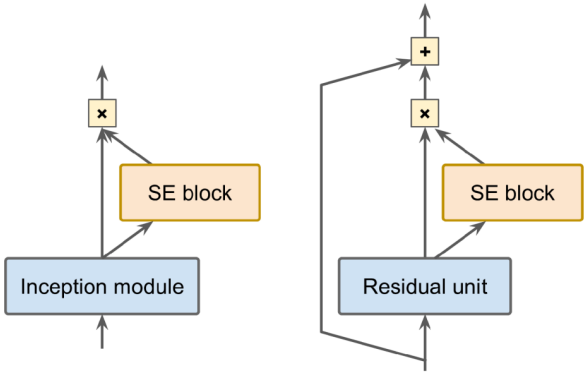

Figure 9: SE-Inception module (left) and SE-ResNet unit (right).

- An SE block is composed of just three layers: a global average pooling layer, a hidden dense layer (**a bottleneck layer**) using the ReLU activation function, and a dense output layer (**a vector with each entry is the weight for each channel**) using the sigmoid activation function.

8. Transfer learning is for **lack of enough data**. It uses trained layers (not updated in the training) and new layers to perform training.

9. Localization.

   (a) Localizing a single object.

   - Easy to design the NN: localizing an object in a picture can be expressed as a **regression task** with 4 dependent variables (horizontal center, vertical center, width, and height) → use MSE loss → better to use IoU (Intersection over Union) accuracy.
   - Hard to get the labels. Crowdsourcing is a solution (Kovashka et al., 2016).

   (b) Classifying and localizing multiple objects: *object detection*.

   - Previous approach: requires running the CNN many times, so it is quite slow.
     - Take a CNN that was trained to classify and locate a single object, then *slide it across the image* → it will detect the same object multiple times at slightly different positions.
     - *Non-max suppression* is used to get rid of such unnecessary duplicating detection by **an objectness score**.
   - Fully convolutional networks (Long et al., 2015): a faster way.
     - Replace the dense layers at the top of a CNN with convolutional layers.

- Take a trained CNN by the previous approach, change the dense layers into convolutional layers with **valid padding**, and **directly copy the weights from the dense layers** to the convolutional layers.
- A Larger image is with larger possible location outputs.

- You Only Look Once (YOLO): is so fast that it can run on a video.
  - It outputs five bounding boxes for each grid cell with five objectness scores.
  - Given a point (x, y), if the center of an object is $(x + \Delta x, y + \Delta y)$ then YOLO predicts the offset $(\Delta x, \Delta y)$.
  - Before training, YOLO finds the sizes of five representative bounding boxes or *anchor boxes* by applying the K-Means algorithm.

10. Semantic segmentation: a more accurate localization task.

- Modified FCN: a trained CNN + convolutional layers replacing the dense layers + an upsampling layer.[6]
  - Upsampling: a *transposed convolutional layer* (stretch the image + a regular convolutional layer).
  - Skip connection: preserve the spatial information (Figure 10).

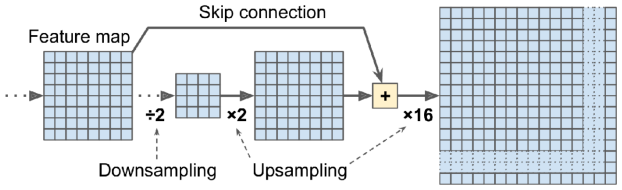

Figure 10: Skip layers recover some spatial resolution from lower layers.

11. Recurrent Neural Networks (RNNs):

- Recurrent neurons and layers (Figure 11).[7] The output of each layer is

$$\boldsymbol{y}_{(t)} = \phi\left(\boldsymbol{w}_x^{\mathrm{T}}\boldsymbol{x}_{(t)} + \boldsymbol{w}_y^{\mathrm{T}}\boldsymbol{y}_{(t-1)} + \boldsymbol{b}\right).$$

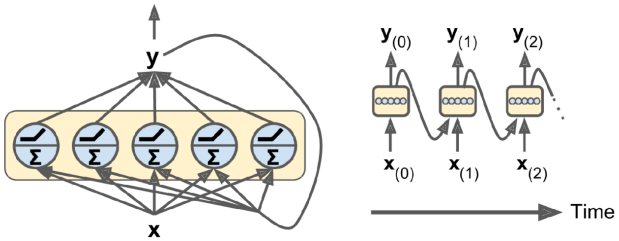

Figure 11: A layer of recurrent neurons (left) unrolled through time (right).

Generally, the message given by a recurrent neuron is the *hidden state* of the neuron, denoted by $\boldsymbol{h}_{(t)} = f(\boldsymbol{h}_{(t-1)}, \boldsymbol{x}_{(t)})$ instead of $\boldsymbol{y}_{(t)}$. Therefore, the general output is

$$\boldsymbol{y}_{(t)} = \phi\left(\boldsymbol{w}_x^{\mathrm{T}}\boldsymbol{x}_{(t)} + \boldsymbol{w}_y^{\mathrm{T}}\boldsymbol{h}_{(t-1)} + \boldsymbol{b}\right).$$

[6] Without the upsampling layer, the last layer outputs feature maps that are smaller than the input image.

[7] A part of a neural network that preserves some state across time steps is called *a memory cell*. The output of a memory cell is a function of historical information.

- Input and output (Figure 12).
  (a) Sequence-to-sequence.
  (b) Sequence-to-vector (only care the last element of the output): *encoder*.
  (c) Vector-to-sequence (input the same vector repeatedly at each time step): *decoder*.
  (d) Encoder-Decoder.

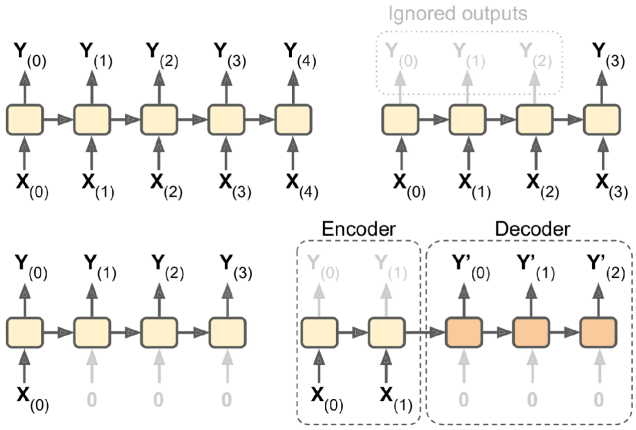

Figure 12: Seq-to-seq (top left), seq-to-vector (top right), vector-to-seq (bottom left), and Encoder–Decoder (bottom right) networks

- Training: *backpropagation through time* (BPTT) first unrolls the RNNs, then takes a copy of RNNs in each time step and performs error backpropagation, then sums over all time steps and completes the update.
- Deep RNNs (Figure 13).

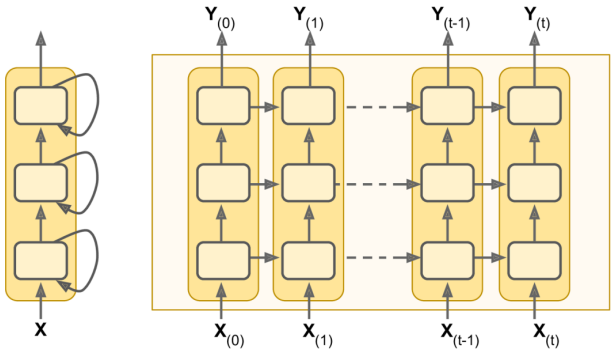

Figure 13: Deep RNN (left) unrolled through time (right).

12. A small clarification.

```
model1= keras.models.Sequential([
            keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
            keras.layers.SimpleRNN(20),
            keras.layers.Dense(1)
            ])
model2 = keras.models.Sequential([
            keras.layers.SimpleRNN(10, return_sequences=True, input_shape=[None, 1]),
            keras.layers.SimpleRNN(30),
            keras.layers.Dense(1)
            ])
```

These two models are different, although the construction is identical in the level of recurrent neurons. It is clear once we catch the point that **in the same layer, all the neurons are working in parallel**. However, neurons in previous layers work first and then the later ones work.

13. Tackling the short-term memory problem.

- Long Short-Term Memory (LSTM, Figure 14) cells (Hochreiter & Schmidhuber, 1997): can learn what to store in the long-term state, what to throw away, and what to read from it by two states and three gates.
  (a) The long-term state $\boldsymbol{c}_{(t-1)}$ traverses the network from left to right, drops some "memory" (controlled by forget gate), and add some other "memory" in the cell (controlled by input gate).
  (b) The short-term (hidden) state $\boldsymbol{h}_t$ gets some long-term information from $\boldsymbol{c}_{(t1)}$ (controlled by output gate).

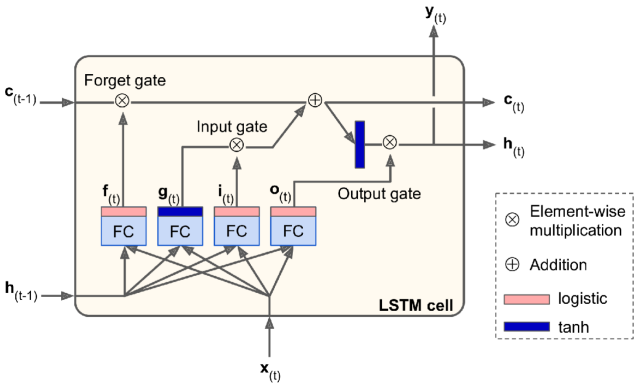

Figure 14: LSTM cell.

- Peephole connections: let the input gate peep the previous long-term state $\boldsymbol{c}_{t-1}$.
- Gated Recurrent Unit (GRU, Figure 15) cells (Cho et al., 2014): a simplified version of the LSTM cell, and it seems to perform just as well.
  - $\boldsymbol{h}_{(t)}$ contains both long-term and short-term information.
  - A single gate controller $\boldsymbol{z}_{(t)}$ controls both the forget gate and the input gate.

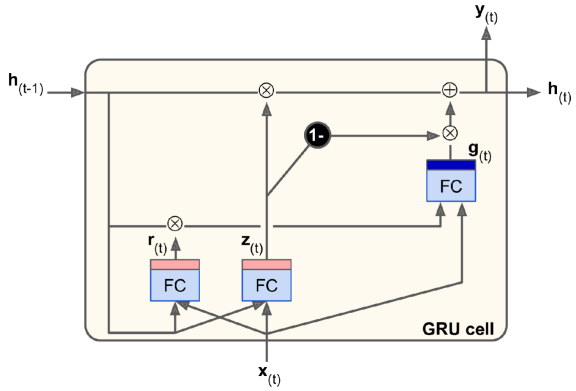

Figure 15: GRU cell.

- Add a 1D convolutional layer to preprocess.
- WaveNet (Oord et al., 2016): stacks 1D convolutional layers, doubling the dilation rate (how spread apart each neuron's inputs are) at every layer → the lower layers learn short-term patterns, while the higher layers learn long-term patterns.

14. Two types of NNs on the natural language processing (NLP) by RNNs.

    (a) A *stateless* RNN learns on random portions of text at each iteration without any information on the rest of the text.

    - Either overlapping or non-overlapping.[8]
    - Can shuffle the dataset.

    (b) A *stateful* RNN preserves the hidden state (the inner value or memory) between training iterations and continues reading, allowing it to learn longer patterns.

    - Non-overlapping.
    - Cannot shuffle.

15. Several stages on NLP.

    - A *character* RNN predicts the next character in a sentence and thus generates some original text.
      - Tokenize each character (including space and punctuation) into an integer.
    - Sentiment analysis.
      - How to tokenize?
        * Tokenize each word into an integer.
        * Tokenize and detokenize text at the subword level in a language-independent way (Kudo & Richardson, 2018; Sennrich et al., 2015; Wu et al., 2016).
      - Masking: tells the model to ignore the padding tokens, so that it can focus on the data.
      - Reusing pre-trained embeddings.
    - Neural machine translation model: encoder-decoder.
      - Traditional RNNs: outputs each word with the highest probability (An example is in Figure 16).
        * Very different length → group sentences into buckets of similar lengths instead of padding all sentences into the same length.
        * Ignore any output past the <eos> token.
        * To avoid time-consuming long vector output, we can use *sampled softmax technique* (Jean et al., 2015).[9]

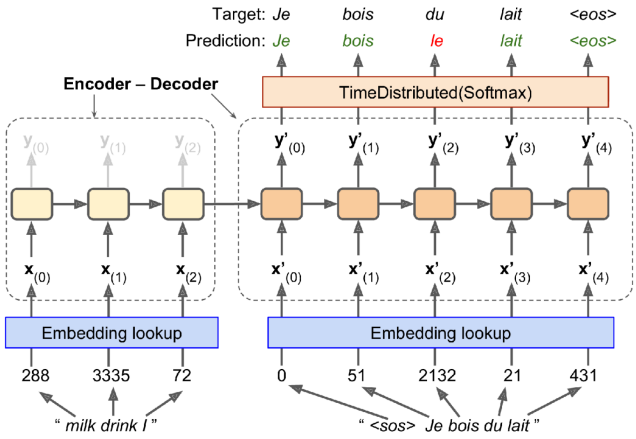

Figure 16: A simple machine translation model.

      - Bidirectional RNNs (Figure 17): The NN can look into the future.

[8] *Truncated backpropagation through time*: cut the long instance over time into shorter (overlapping or non-overlapping) instances.
[9] Look only at the logits output by the model for the correct word and for a random sample of incorrect words

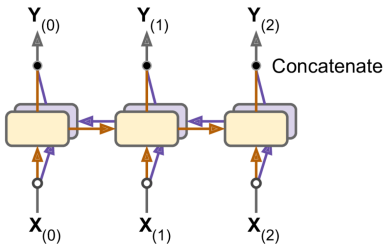

Figure 17: A bidirectional recurrent layer

  - Beam search: gives the model a chance to go back and fix mistakes it made earlier by preserving suboptimal words till it becomes a full sentence then comparing the probability of each candidate sentence. (It works well for short sentences but badly for long sentences due to the limited short-term memory of RNNs).

- Attention mechanisms.
  - Concatenative attention or additive attention (Bahdanau et al., 2014): the decoder will focus on the appropriate words (as encoded by the encoder) at each time step by designating some weights $\alpha_{i,j}$ where **the weights are fitted by a dense NN**(Figure 18).[10]

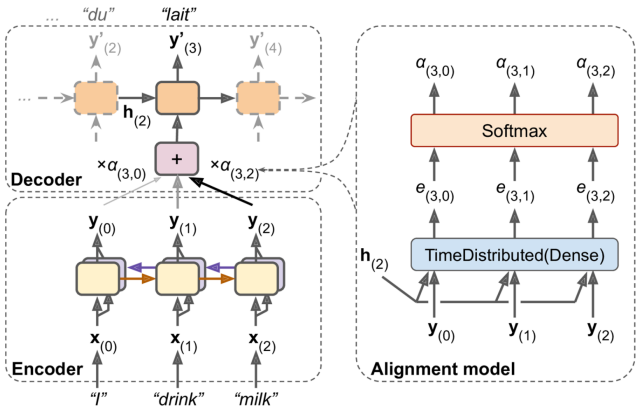

Figure 18: Neural machine translation using an Encoder-Decoder network with an attention model

  - Multiplicative (dot) attention (Luong et al., 2015): the weights are computed by matrix multiplication, which performs better and is much faster than additive attention.
  - General dot product approach (Luong et al., 2015): the encoder outputs first go through a linear transformation (i.e., a time-distributed Dense layer without a bias term) before the dot products are computed.

In summary,

$$\tilde{\boldsymbol{h}}_{(t)} = \sum_i \alpha_{(t,i)} \boldsymbol{y}_{(i)}$$

where $\alpha_{(t,i)} = \frac{\exp\{e_{(t,i)}\}}{\sum_j \exp\{e_{(t,j)}\}}$ with

$$e_{(t,i)} = \begin{cases} \boldsymbol{v}^{\mathrm{T}} \tanh\left(\boldsymbol{W} \begin{bmatrix} \boldsymbol{h}_{(t)} \\ \boldsymbol{y}_{(i)} \end{bmatrix}\right), & \text{additive} \\ \boldsymbol{h}_{(t)}^{\mathrm{T}} \boldsymbol{y}_{(i)}, & \text{dot} \\ \boldsymbol{h}_{(t)}^{\mathrm{T}} \boldsymbol{W} \boldsymbol{y}, & \text{general dot} \end{cases}.$$

[10] Previously, the input is "milk drink I", the output is "Je bois le lait". In this way, there is a long time from input "milk" to the output "lait".

16. More on attention.

- Visual attention: an image → CNN → RNN with an attention mechanism + decoder → a caption.
- The transformer architecture (Vaswani et al., 2023): **without any recurrent or convolutional layers** (Figure 19).

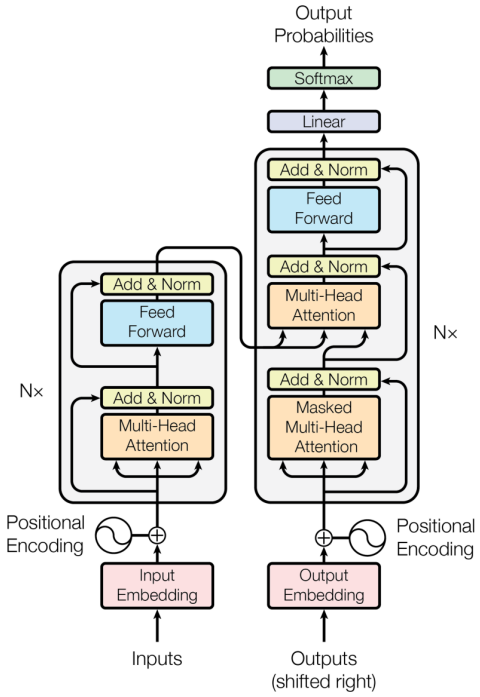

Figure 19: The Transformer architecture.

17. Autoencoder: an encoder (or *recognition network*) that converts the inputs to a latent representation + a decoder (or *generative network*) that converts the internal representation to the outputs.

- PCA with an undercomplete linear autoencoder: to perform simple PCA, we just do not use any activation function. [11]
- Stacked autoencoders or deep autoencoders: add **more layers** and are typically **symmetrical** with regard to the central hidden layer (the coding layer).[12]
- Convolutional autoencoders, e.g., Conv2D and Conv2DTranspose.
- Recurrent autoencoders: encoder (sequence-to-vector RNNs) + decoder (vector-to-sequence RNNs).
- Denoising autoencoders: a regular layer stacked autoencoder with an additional Dropout layer applied to the encoder's inputs. [13]
- Sparse autoencoders: by adding an appropriate term to the cost function, the autoencoder is pushed to reduce the number of active neurons (sparsity) in the coding layer.
  - A simple approach is to use the sigmoid activation function in the coding layer, use a large coding layer, and add some ℓ_1 regularization to the coding layer's activations.
  - Another approach is to measure the actual sparsity of the coding layer at each training iteration and **penalize the model when the measured sparsity differs from a target sparsity**.

[11] Undercomplete means that the internal representation has a lower dimensionality than the input data.

[12] A trick to reduce training time is to tie the weights of the decoder layers to the weights of the encoder layers ($\boldsymbol{W}_{N-L+1} = \boldsymbol{W}_L^{\mathrm{T}}, L = 1, 2, \cdots, N/2$).

[13] The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched-off inputs, just like in dropout. They are both only active during training.

- Variational autoencoders (Kingma & Welling, 2013): *probabilistic* autoencoders[14] + *generative* autoencoders (Figure 20).

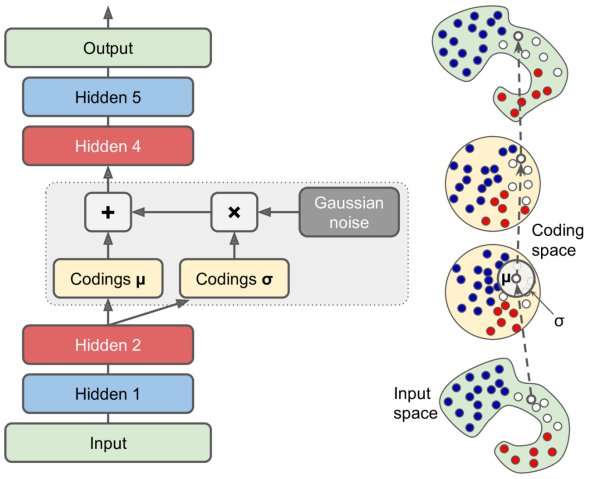

Figure 20: Variational autoencoder (left) and an instance going through it (right)

  - The actual coding is sampled randomly from a Gaussian distribution with mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$.
  - The cost function: usual reconstruction loss + latent loss (pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution: K-L divergence).

18. Generative adversarial network (GAN, Goodfellow et al. 2014): make neural networks compete against each other in the hope that this competition will push them to excel.

    - Components.
      - Generator: Takes a random distribution as input (typically Gaussian) and outputs some images.
      - Discriminator: guess whether the input image is fake or real.
    - Training iteration: first initialize the generator and then perform the following in each iteration.
      - Train the discriminator: input a batch of real images (labeled 1) and an equal number of fake images (labeled 0) produced by the generator and train it by cross-entropy loss.
      - Train the generator: Fronze the weights of the discriminator, use the generator to produce fake images, label them all 1, input to the discriminator, and do the training.
    - Training difficulties: nothing guarantees that the training Nash equilibrium will ever be reached.
      - *Mode collapse*: the generator's outputs gradually become less diverse, making the discriminator less diverse → GANs are very sensitive to the hyperparameters.
      - Solutions: new cost functions, experience replay, and mini-batch discrimination.
    - Other GANs.
      - Deep convolutional GANs (Radford et al., 2016): end up with locally convincing features but overall inconsistencies.
      - Progressive Growing of GANs (Karras et al., 2018): generates small images at the beginning of training, then gradually adds convolutional layers (upsampling layers) to both the generator and the discriminator to produce larger and larger images.[15]

[14] (Here, output is random (as opposed to denoising autoencoders, which use randomness only during training).
[15] This approach resembles greedy layer-wise training of stacked autoencoders.

– StyleGAN (Karras et al., 2019): uses style transfer techniques in the generator to ensure that the generated images have the same local structure as the training images, at every scale,

19. Introduction to reinforcement learning.

- One goal: maximize the lifetime reward.
- Two parties: agent and environment.
- Three elements actions, policy, and rewards (observations).
- Two algorithms: genetic algorithms and policy gradients.

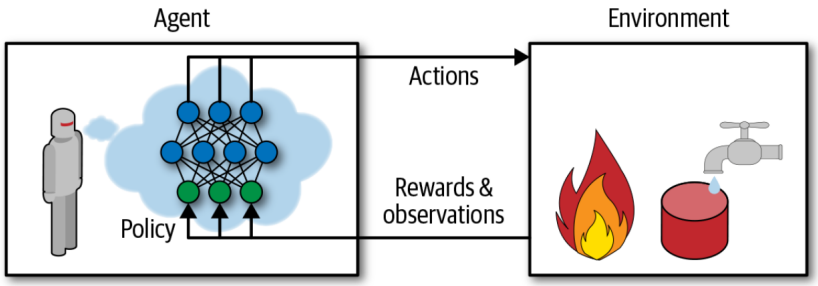

Figure 21: Reinforcement Learning using a neural network policy.

20. REINFORCE algorithms (Williams, 1992): compute the gradients at each step → compute each action's advantage → compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step.

    The learning algorithm for this network is such that at the end of each trial each parameter w_{ij} in the network is incremented by an amount

    $$\Delta w_{ij} = \alpha_{ij}(r - b_{ij})e_{ij}$$

    where α_{ij} is a learning rate factor, b_{ij} is a reinforcement baseline, and $e_{ij} = \frac{\partial \log g_i}{\partial w_{ij}}$ is called the characteristic eligibility (gradient) of w_{ij}.

21. Temporal Difference Learning (TD Learning) algorithm: the agent has only partial knowledge of the MDP.[16]

    The agent uses an exploration policy–for example, a purely random policy–to explore the MDP:[17]

    $$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma V_k(s')) = V_k(s) + \alpha\underbrace{(r + \gamma V_k(s') - V_k(s))}_{\delta_k(s,r,s')}$$

    where α is the learning rate, $t + \gamma V_k(s')$ is called the TD target, and $\delta_k(s, r, s')$ is called the TD error.

22. Q-Learning algorithm: an adaptation of the Q-Value Iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown.[18]

    $$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} Q(s', a').$$

[16] In the value iteration and Q-value iteration, the agent knows full knowledge of the MDP such as the transition probability and reward function.

[17] The TD Learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed.

[18] $a \underset{\alpha}{\leftarrow} b \iff a_{k+1} \underset{\alpha}{\leftarrow} (1-\alpha) \cdot a_k + \alpha \cdot b_k$.

- Q-Learning can work only if the exploration policy explores the MDP **thoroughly enough** since it takes a purely random policy to explore. → A better option is to use ε-greedy policy.[19] It encourages the exploration policy to try new actions. Generally,

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} f\left(Q\left(s',a'\right), N\left(s',a'\right)\right)$$

where $N(s',a')$ is the number of times the action a' was chosen in state s', f is an exploration function, for example, $f(Q,N) = Q + \frac{\kappa}{1+N}$.

- Q-Learning does not scale well to **large (or even medium) MDPs** with many states and actions → A better option is (deep) approximate Q-learning.[20]

$$\underbrace{Q_{\text{target}}(s,a)}_{\text{The training target}} = r + \gamma \cdot \max_{a'} \underbrace{Q_{\boldsymbol{\theta}}(s',a')}_{\text{What DNN gives us}} .$$

Specifically, we generally try to minimize the squared error between the estimated $Q_{\boldsymbol{\theta}}(s,a)$ and $Q_{\text{target}}(s,a)$.

Here are some variants of Deep Q-learning.

(a) Fixed Q-value targets (Mnih et al., 2013): fix the self-supervised problem in the basic deep Q-learning by introducing two models (DQNs).
  - Online model: learn to move the agent around.
  - Target model (just a clone of the online model at regular intervals)[21]: define the targets.

(b) Double DQN (Van Hasselt et al., 2016): fix the overestimating problem of Q-value (caused by noise or uncertainty) in the target network by
  - using the online model when **selecting the best actions** (arg max operation) for the next states,
  - and using the target model only to **estimate the Q-Values** (evaluation) for these best actions.

(c) Importance sampling (IS) or prioritized experience replay (PER) (Schaul et al., 2016): sample important experiences more frequently instead of sampling uniformly.
  - "important": experiences that are likely to lead to fast learning progress.
  - A large TD error $\delta = r + \gamma V(s') - V(s)$ indicates that a transition (s,r,s') is very surprising, and thus probably more informative.
  - Process: record an experience in the buffer, its priority p is set to a very large value → once sampled, set $p = |\delta|$ → The probability P of sampling an experience with priority p is proportional to p^{ζ} where the hyperparameter $\zeta \in [0,1]$ controls the importance level. [22]
  - Bias compensation [23]: designate lower weight during training → set experience's training weight as $w = (nP)^{-\beta}$, where n is the number of experiences in the replay buffer, and the hyperparameter $\beta \in [0,1]$ controls the compensation level.

(d) Dueling DQN (Wang et al., 2016): estimate both the value of the state and the advantage of each possible action. [24]

(e) Rainbow Hessel et al. (2017): combine six different techniques into an agent.

[19] At each step it acts randomly with probability ε, or greedily with probability $1-\varepsilon$.

[20] Find a function $Q_{\boldsymbol{\theta}}(s,a)$ that approximates the $Q(s,a)$ using a manageable number of parameters $\boldsymbol{\theta}$. Mnih et al. (2013) showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering.

[21] The target model is updated much less often than the online model, so the Q-Value targets are more stable.

[22] When $\zeta = 0$, we just get uniform sampling, and when $\zeta = 1$, we get full-blown importance sampling.

[23] Since the samples will be biased toward important experiences, we must compensate for this bias during training in case the model overfits the important experiences.

[24] $Q(s,a) = V(s) + A(s,a)$ where $A(s,a)$ is the advantage of taking the action a in state s. Note that $V(s) = Q(s,a^*)$, then $A(s,a^*) = 0$. So, when computing $\hat{A}(s,a)$ in the model, the dueling DQN subtracts the maximum of advantage $\hat{A}(s,a^*)$.

23. Training and deploying TensorFlow models at scale.

- Manage the trained model.
  - Model Deployment: deploy machine learning models into production environments. This includes running models on real data and creating scripts for regular model execution.
  - Model Serving: wrap models as web services, allowing any part of your infrastructure to query the model through a simple REST API or other protocols.
  - Model Versioning: manage model versions, gracefully transition between models, and roll back to previous versions in case of issues.
  - Model Experimentation: running multiple models in production for A/B testing and experiments.
  - Scaling Services: handle a high volume of query requests and deploy models on cloud platforms like Google Cloud AI Platform, along with monitoring tools.
- Accelerating training: techniques to speed up model training, including GPU and TPU acceleration, as well as training models across multiple devices and servers using TensorFlow's Distribution Strategies API.
  - Designate different GPUs to different .py files (parallel training).
  - Split a GPU into smaller units.
  - Train a model on a single GPU and perform all the preprocessing in parallel on the CPU.
  - Model parallelism and data parallelism.[25]

[25]The model is replicated across every device, and each replica is trained on a subset of the data.

References

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*(61), 2121–2159. Retrieved from http://jmlr.org/papers/v12/duchi11a.html

Gal, Y., & Ghahramani, Z. (2016). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning* (pp. 1050–1059).

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … Bengio, Y. (2014). *Generative adversarial networks.*

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 770–778).

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., … Silver, D. (2017). *Rainbow: Combining improvements in deep reinforcement learning.*

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Jean, S., Cho, K., Memisevic, R., & Bengio, Y. (2015). *On using very large target vocabulary for neural machine translation.*

Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2018). *Progressive growing of gans for improved quality, stability, and variation.*

Karras, T., Laine, S., & Aila, T. (2019). *A style-based generator architecture for generative adversarial networks.*

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kingma, D. P., & Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

Kovashka, A., Russakovsky, O., Fei-Fei, L., Grauman, K., et al. (2016). Crowdsourcing in computer vision. *Foundations and Trends® in computer graphics and Vision*, *10*(3), 177–243.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*.

Kudo, T., & Richardson, J. (2018). Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*.

Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 3431–3440).

Luong, M.-T., Pham, H., & Manning, C. D. (2015). *Effective approaches to attention-based neural machine translation.*

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). *Playing atari with deep reinforcement learning.*

Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., … Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.

Radford, A., Metz, L., & Chintala, S. (2016). *Unsupervised representation learning with deep convolutional generative adversarial networks.*

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016). *Prioritized experience replay.*

Sennrich, R., Haddow, B., & Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909.*

Smith, L. N. (2018). A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820.*

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., … Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 1–9).

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the aaai conference on artificial intelligence* (Vol. 30).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2023). *Attention is all you need.*

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2016). *Dueling network architectures for deep reinforcement learning.*

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, *8*, 229–256.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., … others (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144.*