Multilayer perceptrons

1 Deep learning models layer simple machine learning models and trains the composition jointly, so that earlier layers can learn features which are useful to later layers. The simplest deep learning model is the multilayer perceptron.

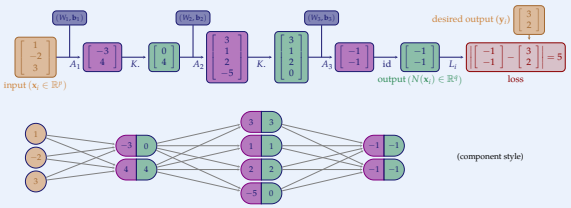
2 A **multilayer perceptron** $N : \mathbb{R}^p \to \mathbb{R}^q$ is a composition of affine transformations and componentwise applications of a function $K : \mathbb{R} \to \mathbb{R}$.

(i) We call K the **activation function**. Common choices:
  (a) $u \mapsto \max(0, u)$ (rectifier, or ReLU)
  (b) $u \mapsto 1/(1+e^{-u})$ (sigmoid)

(ii) *Component-wise application of* K on $\mathbb{R}^t$ refers to the function $K.(x_1, \dots, x_t) = (K(x_1), \dots, K(x_t))$

(iii) An affine transformation from $\mathbb{R}^t$ to $\mathbb{R}^s$ is a map of the form $A(\mathbf{x}) = W\mathbf{x} + \mathbf{b}$, where W is an $s \times t$ matrix and $\mathbf{b} \in \mathbb{R}^s$. Entries of W are called **weights** and entries of $\mathbf{b}$ are called **biases**.

3 The **architecture** of a multilayer perceptron is the sequence of dimensions of the domains and codomains of its affine maps. The neural net represented below has architecture $[3, 2, 4, 2]$.



4 Given training samples $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$, we obtain a neural net regression function by minimizing $L(N) = \sum_{i=1}^{n} L(N(\mathbf{x}_i), \mathbf{y}_i)$ where L is a given function called the **loss** (for example, $L(\mathbf{y}, \mathbf{y}_i) = |\mathbf{y} - \mathbf{y}_i|^2$).

5 For a classification problem with a set $\mathcal{Y}$ of classes, we

(i) let $\mathbf{y}_i = [0, \dots, 0, 1, 0, \dots 0] \in \mathbb{R}^{|\mathcal{Y}|}$, with the location of the nonzero entry indicating class (this is called **one-hot encoding**),

(ii) replace the identity map in the diagram with the **softmax** function $\mathbf{u} \mapsto \left[e^{u_j} / \left(\sum_{k=1}^n e^{u_k}\right)\right]_{j=1}^{|\mathcal{Y}|}$, and

(iii) replace the loss function with $L(\mathbf{y}, \mathbf{y}_i) = -\log(\mathbf{y} \cdot \mathbf{y}_i)$.

6 When the weight matrices are large, they have many parameters to tune. We use a custom optimization scheme:

(i) Start with random weights and a training input $\mathbf{x}_i$.

(ii) **Forward propagation**: apply each successive map and store the vectors at each green or purple node. The vectors stored at the green nodes are called **activations**.

(iii) **Backpropagation**: starting with the *last* green node and working left, compute the change in loss per small change in the vector at each green or purple node. By the chain rule, each such gradient is equal to the gradient computed at right-adjacent node times the derivative of map between the two nodes. The derivative of A_j is W_j, and the derivative of K. is $\mathbf{v} \mapsto \operatorname{diag}\left(\left(\frac{dK}{du}\right)_.(\mathbf{v})\right)$.

(iv) Compute the change in loss per small change in the weights and biases at each blue node. Each such gradient is equal to the gradient stored at the next purple node times the derivative of the intervening affine map. We have $\frac{\partial(W\mathbf{x}+\mathbf{b})}{\partial \mathbf{b}} = I$ and $\mathbf{v}\frac{\partial(W\mathbf{x}+\mathbf{b})}{\partial W} = \mathbf{v}'\mathbf{x}'$. The desired change of each parameter is equal to $-\alpha$ times the derivative of the loss with respect to that parameter, where α is a constant called the **learning rate**.

(v) **Stochastic gradient descent**: repeat (ii)–(iv) for each sample in a randomly chosen subset of the training set (called a **batch**) and determine the average desired change in weights and biases to reduce the loss function. Update the weights and biases accordingly and iterate to convergence.

An **epoch** is an arbitrary number of batches, typically corresponding to one pass through the training data, used to divide time in the training process.

Multilayer perceptrons in tensorflow.keras

1 TensorFlow is the most popular deep learning framework, and Keras provides a convenient interface to TensorFlow.

2 Basic Keras operations:

(i) Store training data as NumPy arrays.

```
import numpy as np
n = 10**4
x_train = np.random.randn(4*n,3)
y_train = np.column_stack((np.sum(x_train,axis=1),
                           np.sum(x_train**2,axis=1)))
x_test = np.random.randn(n,3)
y_test = np.column_stack((np.sum(x_test,axis=1),
                          np.sum(x_test**2,axis=1)))
```

(ii) **Create** the model architecture.

```
import tensorflow as tf
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(units=2,activation='relu',input_dim=3))
model.add(tf.keras.layers.Dense(units=4,activation='relu'))
model.add(tf.keras.layers.Dense(units=2,activation='linear'))
```

(iii) **Compile** (specify the loss function, optimizer, and metrics to track during training):

```
model.compile(loss = 'mean_squared_error',
              optimizer = 'sgd', # stochastic gradient descent
              metrics = ['mean_squared_error'])
```

(iv) **Train** the model.

```
model.fit(x_train, y_train, epochs = 10, batch_size = 100)
```

(v) **Evaluate** the model using the test data.

```
model.evaluate(x_test, y_test)
```

(vi) **Predict** output for a new sample.

```
model.predict(np.expand_dims([1,1,1],0))
```

(vii) **Save** the model.

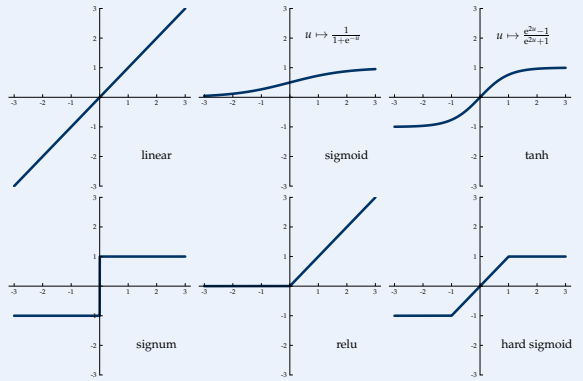
```
model.save('my-model.h5')
# to reload:
model = tf.keras.models.load_model('my-model.h5')
```

A neural network timeline

1943 McCulloch and Pitts proposed the first mathematical model of the neuron.
1950 Minsky and Edmonds build the first neural network machine (SNARC).
1958 Rosenblatt proposed the single-layer perceptron.
1961 Rosenblatt proposed the multi-layer perceptron.
1969 Minsky and Papert's book *Perceptrons* led to the first AI winter.
1970 Seppo Linnainmaa introduced the general backpropagation algorithm.
1974 Paul Werbos applied backpropagation to neural networks.
1986 Backpropagation applied and popularized by Rumelhart, Hinton, and Williams.
1987 The first International Conference on Neural Networks.
1989 Cybenko showed that neural networks can approximate arbitrary continuous functions.
2006 Hinton introduced the term "deep learning".
2010 Fei-Fei Li's team at Stanford built the ImageNet database of millions of labeled real-world images.
2012 Krizhevsky, Sutskever, and Hinton achieved breakthrough improvement on ImageNet.
2014 Facebook's system DeepFace achieves human-level facial recognition.
2016 Google DeepMind's AlphaGo defeated a human world champion Go player.
2018 Google DeepMind's AlphaZero, trained entirely on self-play, defeated the best conventional chess engine
2019 OpenAI's GPT-2 generates intelligible paragraph responses to natural language prompts

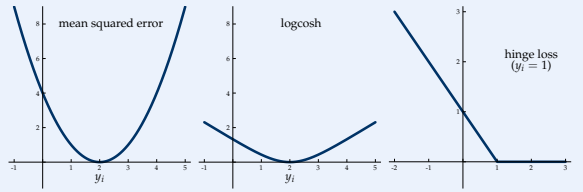
Activations and loss functions

1 Common activations:



2 Choice of loss function depends on the problem type (regression or classification) and how the neural net output is interpreted. For example, if the last layer outputs a **probability vector** (a vector with nonnegative entries which sum to 1), we would use cross entropy.

name	formula	output $\mathbf{y}$	target $\hat{\mathbf{y}}$
mean squared error	$\lvert\mathbf{y} - \hat{\mathbf{y}}\rvert^2$	vector or scalar	vector or scalar
logcosh	$\log\cosh(y - \hat{y})$	scalar	scalar
cosine proximity	$\frac{\mathbf{y}\cdot\hat{\mathbf{y}}}{\lvert\mathbf{y}\rvert\lvert\hat{\mathbf{y}}\rvert}$	vector	vector
cross entropy	$-\log(\mathbf{y} \cdot \hat{\mathbf{y}})$	prob. vector	indicator vector
hinge loss	$\max(0, 1 - y\hat{y})$	scalar	binary ($\{-1, +1\}$)



Training techniques

1 Neural networks often have many parameters and may **overfit** the training data.

(i) (ℓ^2) **Regularization**. Penalize large parameters by adding a multiple (λ) of the sum of the squares of the parameters to the objective function being minimized. This has the effect of changing the update rule for each weight from $w \leftarrow w - \alpha\frac{\partial L}{\partial w}$ to $w \leftarrow w(1 - 2\alpha\lambda) - \alpha\frac{\partial L}{\partial w}$, where α is the learning rate and L is the loss.

(ii) **Architecture**. Build the architecture to reflect an domain-specific knowledge. Examples: convolutional neural networks for image data, recurrent neural networks for natural language data. Favor deep nets over wide ones.

(iii) **Early stopping**. Halt the optimization process early, for example when the error on a witheld portion of the training data begins to rise.

(iv) **Dropout**. Remove each neuron independently with probability p (usually between $\frac{1}{5}$ and $\frac{1}{2}$) and train the reduced network on a mini-batch of samples. Repeat with a freshly sampled subset on the next mini-batch.

2 Updates for early layers can be very small or very large for a deep network, because the corresponding gradients are obtained by multiplying many matrices (the **vanishing/exploding gradient problem**).

(i) Use ReLU instead of sigmoid activation, since its derivative is 1 on the positive real line instead of being no more than $\frac{1}{4}$.

(ii) Greedily pre-train the network layer-by-layer using unsupervised learning.

Convolutional neural networks

1 The convolutional neural network architecture is designed for image data. Rather than flattening the pixel values of the $n \times n$ image into a vector, we preserve its two-dimensional structure and treat the input layer as a matrix (or a rank-3 tensor if the image has multiple *channels*, such as red-green-blue).

2 A **stencil** is a square subgrid of a given image.

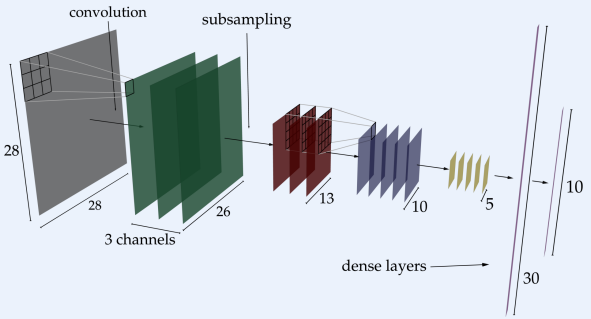
(i) The set of $s \times s$ stencils in an $n \times n$ image form a $(n-s+1) \times (n-s+1)$ grid.

(ii) Partitioning an $n \times n$ grid into $s \times s$ stencils yields a $(n/s) \times (n/s)$ grid.

3 **Convolutional layer**. We fix a stencil size s and define the second layer to be a tensor whose first two dimensions correspond to the grid of $s \times s$ stencils in the first layer and whose thickness t is arbitrary. Each image in the second layer is associated with an $s \times s \times t$ tensor called a *filter* and a scalar called the *bias*. The pixel values for that image are obtained by dotting the filter with the corresponding stencil in the previous layer and adding the bias. The thickness of the second layer is chosen as a part of the architecture, and the filter entries and biases are parameters to be learned.

4 **Subsampling layer** The transformation from the second layer to the third layer applies a fixed function to each stencil in a partition of each image. This is called **max pooling** if the function is `max`).

5 A typical convolutional architecture consists of dense layers as well as alternating convolutional and subsampling layers.



6 CNNs in Keras:

```
from tensorflow.keras import layers
from tensorflow.keras import models

model = models.Sequential([
  layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.Flatten(),
  layers.Dense(64, activation='relu'),
  layers.Dense(10, activation='softmax')
])
```

Training data should be a 4-dimensional `ndarray` with dimensions

$$\text{samples} \times \text{height} \times \text{width} \times \text{channels}$$

Transfer learning

1 **Transfer learning** entails re-using some of the layers of a network which has been trained on a general data set (like ImageNet). Pre-trained classification models available in Keras include `Xception`, `InceptionV3`, `ResNet50`, `VGG16`, `VGG19`, and `MobileNet`.

2 Transfer learning in Keras:

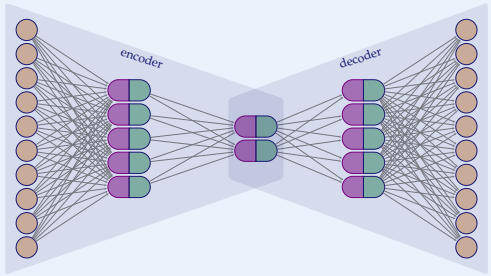
```
from tensorflow.keras.applications import VGG16
from tensorflow.keras import layers
from tensorflow.keras import models

conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(56,56,3))
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
conv_base.trainable = False
```

Autoencoders

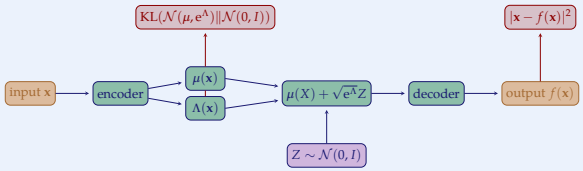
1 An autoencoder is a neural network which is trained to approximate the identity function. One of the hidden layers is identified as the **encoded layer** or **bottleneck**.

2 The map from the input to the encoded layer is called the **encoder**, and the map from the encoded layer to the output is the **decoder**.



3 Encoding an input sample provides a lower-dimensional representation of the sample from which it may be approximately reconstructed using the decoder.

4 A *variational* autoencoder architecture adds random noise at the encoding layer. The encoder outputs two vectors, one of which serves as the mean and the other as the diagonal log-covariance matrix for a multivariate Gaussian which is fed into the decoding layers.



5 The variational autoencoder loss function penalizes deviation of μ and Λ from zero using the KL divergence

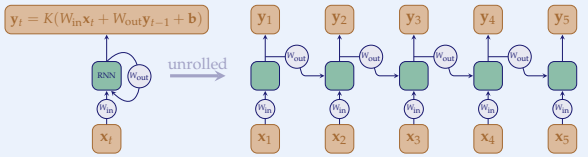
$$\mathrm{KL}\left(\mathcal{N}(\mu,\Sigma) \| \mathcal{N}(0,I)\right) = -\frac{1}{2}\left(\log\det\Sigma - \mathrm{trace}(\Sigma) - |\mu|^2 + n\right).$$

6 A variational autoencoder can be used to generate new samples by applying the decoder to a standard multivariate Gaussian.

Recurrent neural networks

1 **Recurrent** architectures are designed for sequential data and have cycles in the connections between neurons. Input data are fed into the network sequentially, and the cycles allow the calculations at each time step to incorporate computed data from previous time steps.

2 Given matrices W_{in} and W_{out} and a bias term $\mathbf{b}$, a recurrent network with a single `SimpleRNN` cell with activation K outputs $\mathbf{y}_t = K(W_{\text{in}}\mathbf{x}_t + W_{\text{out}}\mathbf{y}_{t-1} + \mathbf{b})$ at time t, where $\mathbf{x}_t$ is the time-t input value. *Example. let $\mathbf{x}_t$ be the tth word in a sentence, and let $\mathbf{y}_t$ be a prediction of the $(t+1)$st word in the sentence.*



3 RNNs in Keras:

```
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential([
  layers.SimpleRNN(50, input_shape = (28,28)),
  layers.Dense(10, activation='softmax')
])
```

The input shape tuple for SimpleRNN should specify the number of time steps and the dimension of the vector input at each time step, respectively.

Neural networks and other machine learning models

1 Popular machine learning models can be realized using neural architectures. Conversely, neural networks may be viewed as traditional machine learning models *made composable*.

(i) The support vector machine is a one-layer neural network with hinge loss, linear activation, and L^2 regularization.

(ii) Linear regression is a one-layer neural network with linear activation and mean squared error.

(iii) Logistic regression is a one-layer neural network with sigmoid activation and cross-entropy loss.

(iv) The singular value decomposition computes the best rank-k approximation of a given matrix ($U\Sigma_k V'$, where Σ_k is obtained from Σ by replacing all but the first k singular values with zero), and so does an autoencoder with linear activation and mean squared error.

Universal approximation and interpretability

1 The **universal approximation theorem** (UAT). If K is a bounded activation or ReLU, then a multilayer perceptron with activation K can represent any continuous input-output relationship arbitrarily well: if $f : D \to \mathbb{R}^m$ is a continuous function and $D \subset \mathbb{R}^n$ is closed and bounded, then for any $\epsilon > 0$, there exists an alternating composition of affine transformations and pointwise applications of K which approximates f within a tolerance of ϵ at every point in D.

2 The idea for proving UAT (in the $\mathbb{R}^1$ to $\mathbb{R}^1$ case, with sigmoid activation) is to use two neurons in a single hidden layer to build a function which approximates the indicator function of an given interval. Then more pairs of neurons are added in the middle layer to obtain approximate indicators of other intervals. These can be scaled and added to represent any step function (⎍).

3 UAT does not address *learnability*: whether it is feasible to train a neural net to approximate a given input-output relationship is a separate question.

4 Deep learning **interpretability** is human understanding of how a neural net works. Interpretation techniques include (1) attribution (ascertaining which components of a given input vector are most important for determining its output) and (2) feature visualization (visualizing the role of a single neuron or set of neurons in a network).

5 **Attribution**: compute the derivative of the desired neuron's activation with respect to the input. (*Example: derivative of pre-softmax "labrador retriever" output neuron for VGG16 network*)

6 **Feature visualization**.

(i) Select a neuron (or layer or channel) and identify the images in the training or test set which maximizes its activation.

(ii) Select a neuron (or layer or channel) and use backpropagation and gradient ascent starting from an image of random pixel values to generate an image which maximizes its activation.

Batch normalization

1 A **batch normalization** layer has the same number of neurons as the preceding layer, with each neuron connected to a single neuron in the preceding layer. It scales and shifts the incoming activations on a neuron-by-neuron basis so as to achieve a given sample mean β and sample standard deviation γ across the samples in a given mini-batch.

2 If $\mathbf{x}$ is a vector of incoming activations at a particular neuron (with length equal to the number of samples per mini-batch), then the output value at that neuron is:

```
μ̂ = mean(x)
σ̂² = std(x)
x_out = β + γ*(x-μ̂)/√(σ̂²+ε)
```

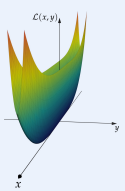
3 The parameters β and γ are learned during training. Therefore, a batch normalization layer with n units has $2n$ trainable parameters.

4 The activation mean and standard deviation estimates used to standardize the activations may be computed with **momentum** $\alpha \in (0,1)$, as an exponential moving average of current and preceding estimates. Each mean estimate may be computed as a weighted average of the previous mean estimate and the new mini-batch mean, with weights α and $1-\alpha$, respectively. Similarly for the standard deviation estimate.

5 Batch Normalization in Keras: `layers.BatchNormalization()`

Optimizers

In the algorithms below, $\mathcal{L}$ represents an objective function to be minimized, `m` represents the current estimate of the gradient, `α` represents a momentum parameter between 0 and 1, and `w` represents the current weight vector (the set of parameters describing the model). The update rule for each iterative algorithm is shown in pseudocode.



1 **Gradient descent with momentum**. Use exponential moving average to retain preceding gradient estimates.

```
m = α*m + (1-α)*∇ℒ(w) # exponential moving average
w -= η*m # update weight
```

2 **Nesterov accelerated gradient**. Look ahead slightly in estimating the gradient.

```
m = α*m + (1-α)*∇ℒ(w-α/(1-α)*η*m) # estimate gradient slightly ahead
w -= η*m # update weight
```

3 **AdaGrad**. Normalize componentwise based on accumulated squared derivative (`.^` and `./` mean componentwise exponentiation and division). Helps with pathological curvature (figure above).

```
s += ∇ℒ(w).^2
w -= η*∇ℒ(w)./√(s+ε)
```

4 **RMSProp**. AdaGrad but with exponential moving average instead of sum.

```
s = α*s + (1-α)*∇ℒ(w).^2
w -= η*∇ℒ(w)./√(s+ε)
```

5 **Adam**. Momentum plus RMSProp.

```
# exponential moving averages for gradient and squared gradient
m = α₁*m + (1-α₁)*∇ℒ(w)
s = α₂*s + (1-α₂)*∇ℒ(w).^2

# give m and s a little boost early in training:
α₁ᵗ *= α₁
α₂ᵗ *= α₂
m̂ = m/(1-α₁ᵗ)
ŝ = s/(1-α₂ᵗ)

# apply AdaGrad scaling to the momentum gradient estimate
w -= η*m̂./√(ŝ+ε)
```

Neural networks and computation

1 **Processing hardware**

(i) The **CPU** (central processing unit, mostly 64-bit) is the primary component in a computer that executes program instructions.

(ii) **GPU**s (graphics processing units, 32-bit) are specialized chips designed to perform graphics-related matrix calculations in parallel.

(iii) **TPU**s (tensor processing units, as little as 8-bit) are specialized chips designed to perform machine learning calculations.

2 **Parallelization**

(i) **Hyperparameter parallelization**. Try various choices of model hyperparameters simultaneously.

(ii) **Model parallelization**. We can split each layer into multiple pieces; for example, in a convolutional neural net we could put half the channels in a given layer on one GPU and half on another.

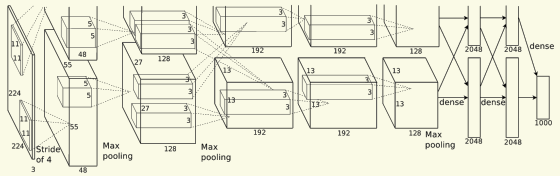
(iii) **Data parallelization**. We can train our model on separate mini-batches simultaneously.

3 **Model Compression**. Deploying a neural network on limited hardware might require shinking the model. One strategy involves training a smaller neural network (a **mimic model**) on the outputs of a larger model trained on the original data.

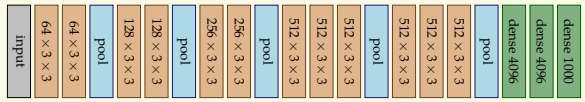
Convolutional architecture case studies

1 **LeNet** (1998). A 6-channel convolution+pool, a 16-channel convolution+pool, a 120-unit fully connected layer, an 84-unit fully connected layer, and a 10-unit output layer (for classifying MNIST digits).

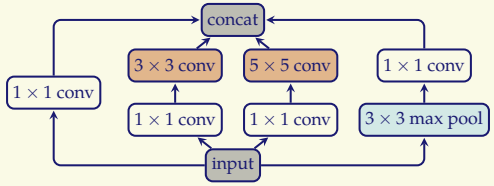
2 **AlexNet** (2012). Much larger convolutional neural network, with channels for each layer split across two GPUs. ImageNet winner.



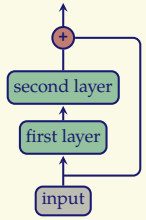
3 **VGG** (2014). Deeper and with smaller filters than AlexNet.



4 **GoogLeNet** (2014). Based around the **inception** module, a parallel collection of convolutions of different sizes. Also very deep; reduced unit-expensive dense layers and injected gradient into earlier layers using two auxiliary classification stacks. ImageNet winner.



5 **ResNet** (2015). Even deeper; mitigates vanishing gradient using residual connection which adds input to layer output (so layers learn diffs). ImageNet winner.



Natural language processing

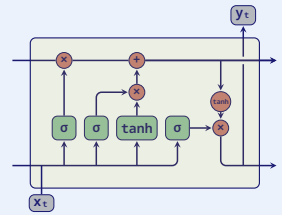
1 Using neural networks on language data requires encoding it in vector form. For example, each word can be associated with a vector, making a document a sequence of vectors.

2 Given a corpus of text, the **word2vec** embedding trains a two-layer, linear-activation/softmax-activation, bias-free neural network to map one-hot encoded words to vectors which have high values in the positions of the words which appear within a fixed-size window around the input word. The loss for a given sample is the sum of the logs of the reciprocals of the values in the positions of the context words.

3 **Negative sampling** is a technique for accelerating training: we perform the softmax calculation on the context word positions and a random subset of the remaining positions.

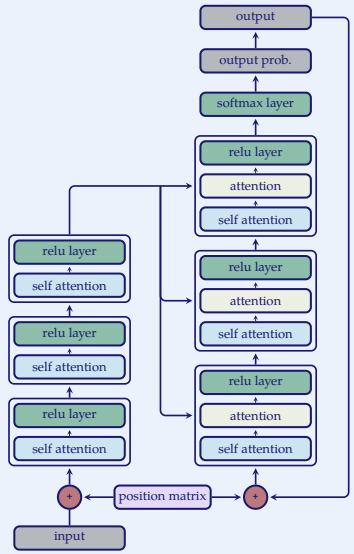
4 word2vec's advantages: (1) words are represented in a vector space of modest dimension, and (2) some aspects of language meaning have vector space analogues. For example, synonyms are often nearest neighbors, and the analogy $a : b :: c : d$ can be completed by finding the value of d which most nearly satisfies the equation $a - b = c - d$.

5 To model the sequential aspect of the textual data, we can use a recurrent neural net. However, SimpleRNN cells don't effectively learn relationships involving long-term memory. The **LSTM cell** is used instead (green boxes represent dense, trainable layers; labels indicate activations):



The top line is for carrying long-term information. The multiplication operation is for deletion, and the sum for addition.

6 BERT and GPT-2 are modern neural network architectures with excellent results on sequence-to-sequence tasks (like sentence translation). They are based on a mechanism called a **transformer**, which *encodes* the input and then *decodes* it to produce an output.



7 **Attention** applies weights to the elements of the input sequence to focus on the most relevant ones (like when you're translating a sentence and look especially at the word `turtle` when you write `дубора`). **Self-attention** focuses on specific words during the encoding process, like when you look at *frisbee* when interpreting *it* in the sentence *She caught the frisbee and threw it*. The output of a self-attention layer is defined to be

$$\text{concat}(\text{head}_1, \dots, \text{head}_n)W^O,$$

where

$$\text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

for each $i \in \{1, 2, \dots, n\}$, and where

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

The entries of the matrices (W_i^Q, W_i^K, W_i^V) and W^O are learned during training.

8 Attention layers are similar but with

$$\text{head}_i = \text{Attention}(XW_i^Q, ZW_i^K, ZW_i^V),$$

where Z is the matrix of outputs from the encoder stack.

9 Probabilities for the target vocabulary are output by a dense layer with softmax activation. An output word is chosen, and this word as well as previous ones are fed back into the decoder stack as input to determine the next word. The process ends when the decoder outputs a special `end` token.

10 Transformers account for word order by adding pre-determined position-indicating vectors to each word. Also, each layer's output is added to its input (residual connection) and z-scored (**layer normalization**) before forwarding to the next layer.

Reinforcement Learning

1 An RL problem consists of an **agent** sequentially interacting with an **environment** and receiving **rewards** for its actions. The goal of the agent is to maximize reward.

2 The agent will typically attempt to make good decisions, but occasionally making other decisions will help the agent better learn the environment (the **exploration-exploitation** tradeoff).

3 RL problems are mathematically formalized as **Markov Decision Processes** (MDPs). An MDP is the collection $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$, where

(i) $\mathcal{S}$ is the set of possible states of the environment,

(ii) $\mathcal{A}$ is the set of possible actions the agent can take,

(iii) $\mathcal{R}$ is the function which specifies the distribution of the reward values the agent will receive for each state-action pair,

(iv) $\mathbb{P}$ is the function which gives—for each state-action pair—the distribution on $\mathcal{S}$ of the next-state value returned by the environment, and

(v) γ is a factor used to discount future rewards.

4 Starting from $s_0 \in \mathcal{S}$, the agent chooses an action a_0, receives a reward r_0 and a next-state value s_1 from the environment (sampled from the distributions $\mathcal{R}_{(s_0,a_0)}$ and $\mathcal{P}_{(s_0,a_0)}$), chooses another action a_1, and so on. The resulting **trajectory** is written as $\tau = (s_0, a_0, r_0, s_1, a_1, \dots, s_{\texttt{done}})$, where `done` is the index where the environment returns a terminal state.

5 The agent chooses its actions using a **policy**, which is a function π from the state space to the action space. The optimal policy is the one which maximizes the expected accumulated discounted reward $\sum_{t\geq 0} \gamma^t r_t$.

6 The **Q-value** of a state-action pair is the expected accumulated discounted reward starting from that state and action. **Q-learning** involves learning the function from state-action pairs to their Q-values.

7 The **Bellman equation** says that the Q-value for a state-action pair is the expected reward for that state-action pair plus the discounted expected Q-value of the next state and the best action for that state.

$$Q(s, a) = \mathbb{E}_{s' \sim \mathbb{P}_{s,a}}[r + \gamma \max_{a'} Q(s', a')].$$

8 The agent can learn Q-values by *bootstrapping*: beginning from some random initial guesses for the Q-values, we update the value of $Q(s, a)$ using the right-hand side of the Bellman equation. This procedure converges to the true Q-function under some technical hypotheses.

9 If the state-action space is large, the agent may learn the Q-function using a function approximator like a neural network. The neural net is iteratively fit to the right-hand side of the Bellman equation as the agent observes the states and rewards returned by the environment.

10 The agent may alternatively iterate directly on its policy function (**policy gradient** methods). Given a policy π defined in terms of some parameters θ, we define the expected accumulated reward for an agent following that policy:

$$J(s) = \mathbb{E}[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots]$$

11 The agent will adjust its weights using gradient ascent with some learning rate α. $\theta \Leftarrow \theta + \alpha \frac{\partial J}{\partial \theta}$.

12 The best parameter update direction can be approximated using Monte Carlo

$$\overline{\sum_{\tau}} R(\tau) \sum_{t\geq 0} \frac{\partial}{\partial \theta} \log \pi_\theta(s_t, a_t),$$

where $\overline{\sum_\tau}$ indicates taking a mean over a collection of sample trajectories τ. The derivatives $\frac{\partial}{\partial \theta} \log \pi_\theta(s_t, a_t)$ may obtained directly from the model (e.g., backprop for a neural net).

13 Policy-gradient methods and Q-learning can be combined (**actor-critic** models). The expected reward values are approximated using the critic model, and the critic model in turn is trained on the state-action pairs visited by the actor.

Big Data

1 **Big data** refers to processing of datasets which are too large to fit into RAM on a single machine. Big datasets may be partitioned and stored on a connected collection of computers called a **cluster**.

2 **MapReduce** is a model for handling a particular class of computation on distributed datasets. A mapreduce operation consists of *mapping* a function over the elements in the dataset and repeatedly applying a binary operation to *reduce* the data. Such an operation may be performed by each node in the cluster on its own dataset, and the results from the nodes may then be collected and reduced. **Hadoop** is an influential implementation of the MapReduce model.

3 **Spark** is an alternative to MapReduce which is more faster and more flexible. Spark is written in Java but can be accessed from Python using `pyspark`.

4 The core data structure in Spark is the **RDD**, or resilient distributed dataset. An RDD is a container which presents similarly to a list in the Python environment but which can store data in (and dispatch computations to) a cluster. RDDs can store arbitrary Python objects.

5 `SparkContext` is a Python class which provides a low-level interface to a Spark instance. It can be used to create or access an RDD.

6 `SparkSession` is a Python class which provides a higher-level interface to RDDs which store rows of a table. It allows you to interact with a tabular RDD using Pandas-like methods or using SQL queries.