CSE428: Image Processing

Lecture 15

CNN Training & Applications

Glossary

- **Sample or input**—One data point that goes into your model.
- **Mini-batch** or **batch**—A small set of samples (typically between 8 and 128) that are processed simultaneously by the model. The number of samples is often a power of 2, to facilitate memory allocation on GPU. When training, a mini-batch is used to compute a single gradient-descent update applied to the weights of the model.

Glossary

- **Prediction** or **output**—What comes out of your model.
- **Target**—The truth. What your model should ideally have predicted, according to an external source of data.
- **Prediction error** or **loss value**—A measure of the distance between your model's prediction and the target.

Glossary

- **Classes**—A set of possible labels to choose from in a classification problem. For example, when classifying cat and dog pictures, “dog” and “cat” are the two classes.
- **Label**—A specific instance of a class annotation in a classification problem. For instance, if picture #1234 is annotated as containing the class “dog,” then “dog” is a label of picture #1234.
- **Ground-truth** or **annotations**—All targets for a dataset, typically collected by Humans.

Glossary

- **Binary classification**—A classification task where each input sample should be categorized into two exclusive categories.
- **Multiclass classification**—A classification task where each input sample should be categorized into more than two categories: for instance, classifying handwritten digits.
- **Multilabel classification**—A classification task where each input sample can be assigned multiple labels. For instance, a given image may contain both a cat and a dog and should be annotated both with the “cat” label and the “dog” label. The number of labels per image is usually variable.

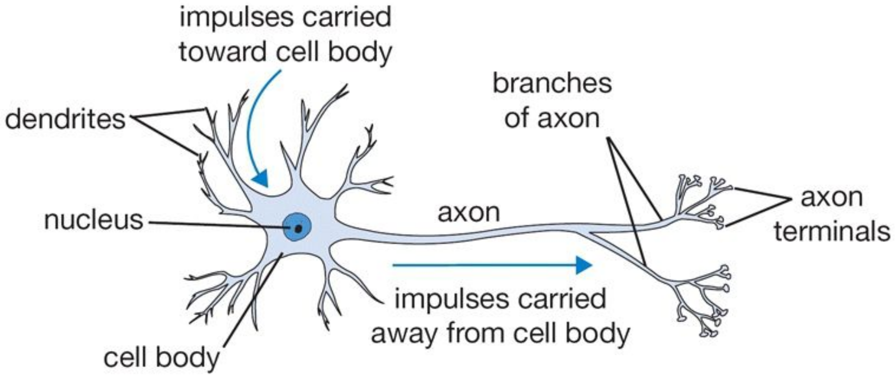
Glossary

- **Scalar regression**—A task where the target is a continuous scalar value. Predicting house prices is a good example: the different target prices form a continuous space.
- **Vector regression**—A task where the target is a set of continuous values: for example, a continuous vector. If you're doing regression against multiple values (such as the coordinates of a bounding box in an image), then you're doing vector regression.

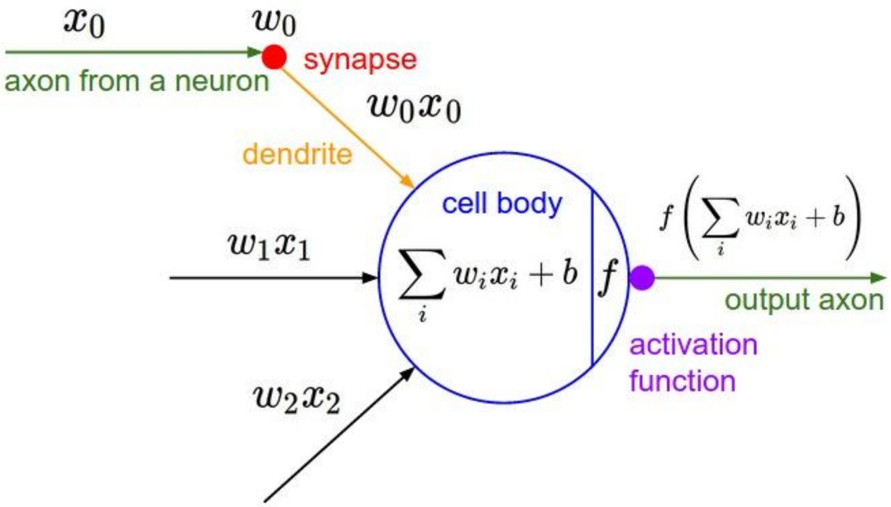
Contents

Activation Function

The activation function: introduces nonlinearity in computation!

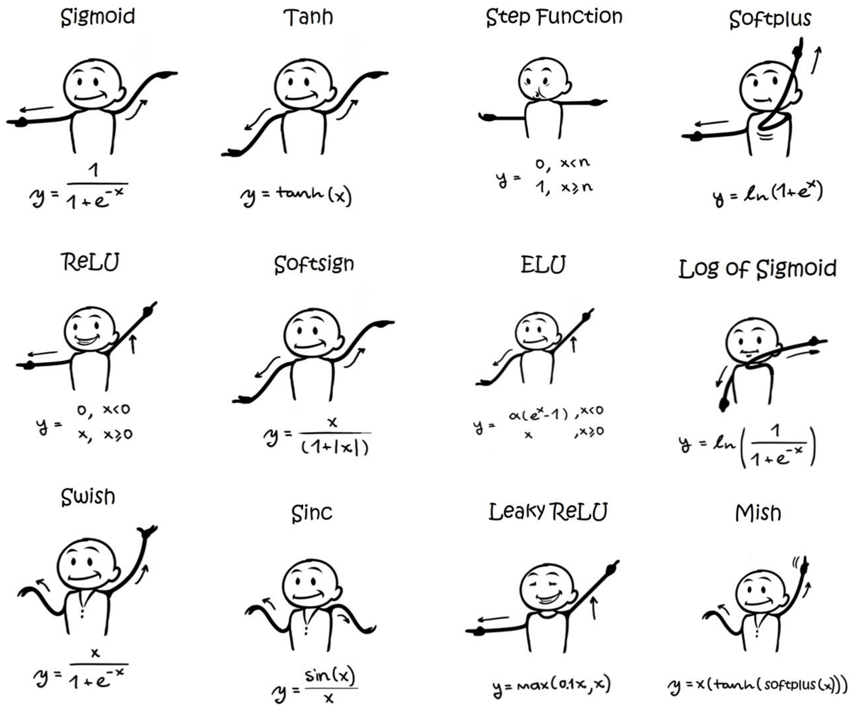


Biological Neuron



Perceptron
(mathematical model of a biological neuron)

Activation Functions



https://twitter.com/AnkitKalauni/status/1464927175996411908

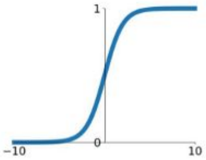
Activation Functions

Activation function: many choices, each with their unique advantage and disadvantages. Common choices are:

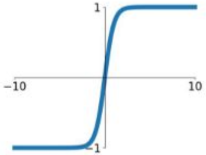
- Sigmoid
- tanh
- Maxout
  - ReLU
  - Leaky ReLU
- ELU

ReLU is mostly used

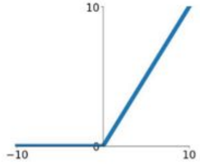
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



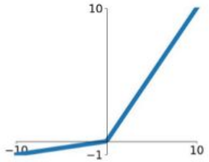
tanh
$\tanh(x)$



ReLU
$\max(0, x)$

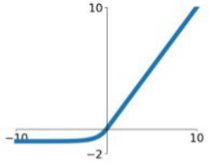


Leaky ReLU
$\max(0.1x, x)$



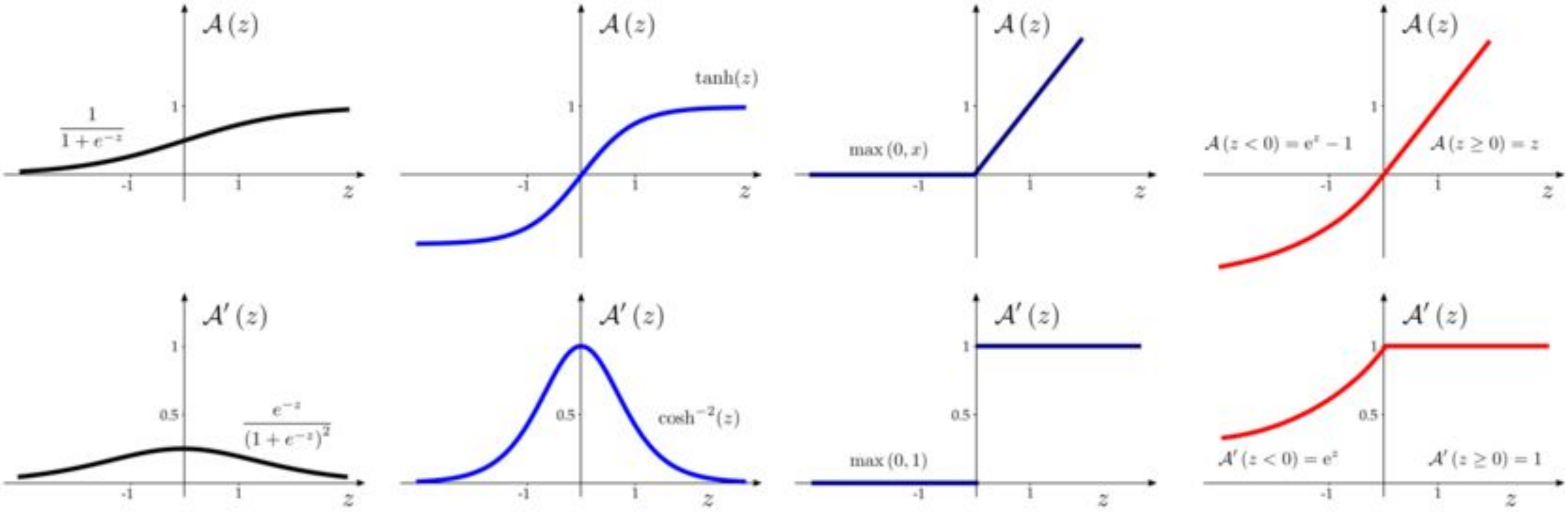
Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

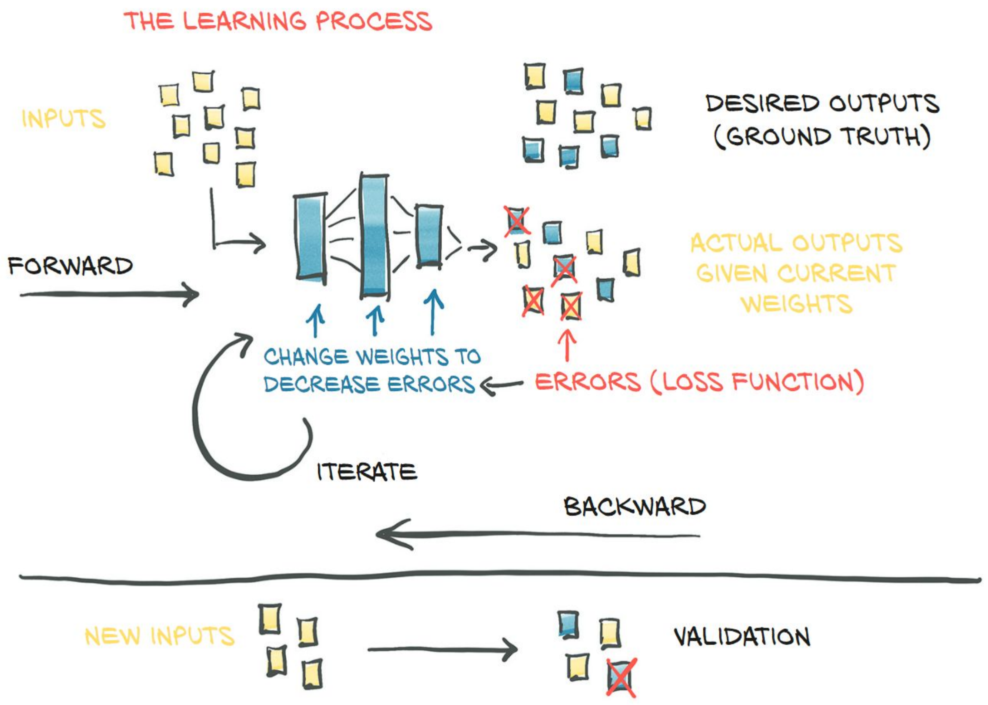


Activation Functions

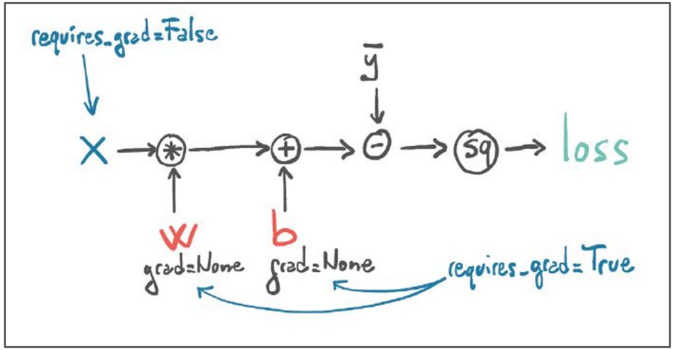
Derivatives are very important for learning & backpropagation, so avoid using activation functions which produce 0 gradients (vanishing gradient problem)

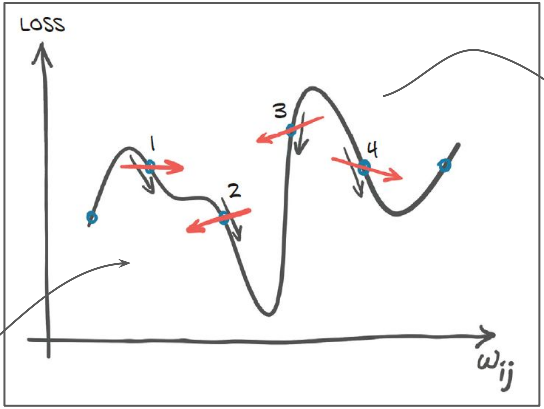


Deep Learning Pipeline

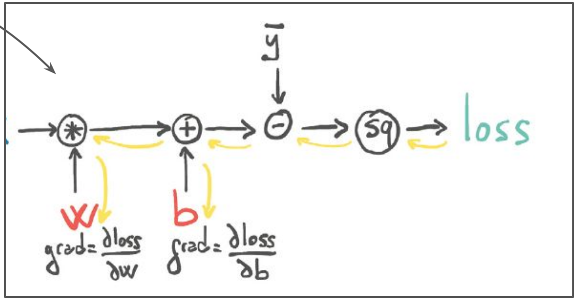


Forward and Backward Propagation





Back-prop: calculate the gradient (chain rule of differentiation)



loss $\mathcal{L}(m_{w,b}(x))$

$$\nabla_{w,b}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial w}, \frac{\partial \mathcal{L}}{\partial b}\right) = \left(\frac{\partial \mathcal{L}}{\partial m}\cdot\frac{\partial m}{\partial w}, \frac{\partial \mathcal{L}}{\partial m}\cdot\frac{\partial m}{\partial b}\right)$$

gradient

partial derivatives

model $m_{w,b}(x)$

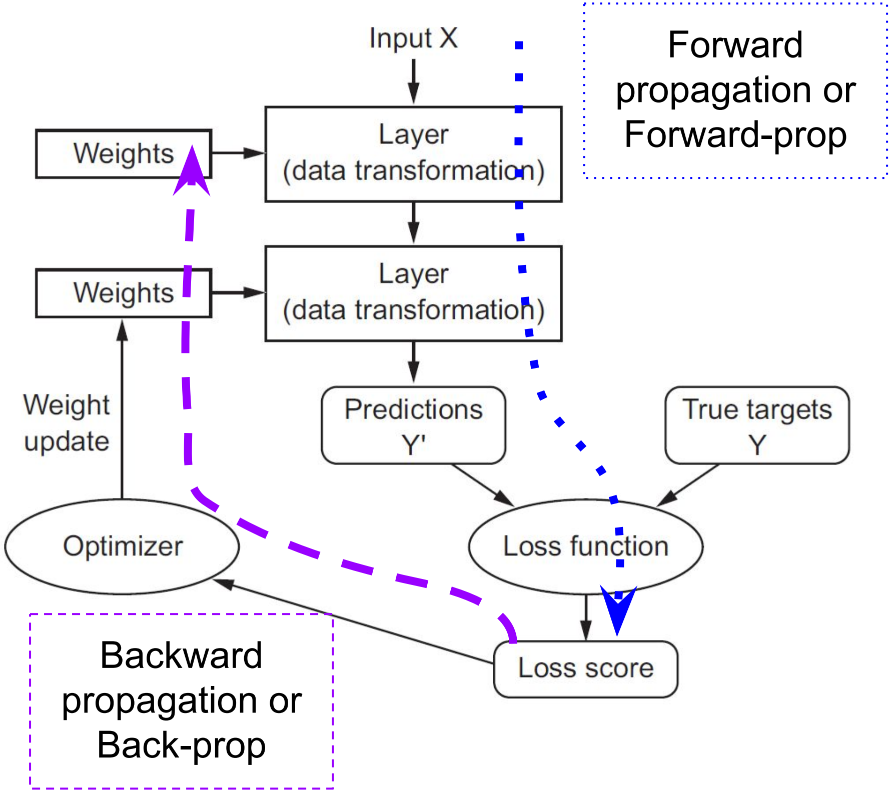
parameters

Forward-prop: calculate the loss

Deep Learning Pipeline

Deep Learning Pipeline

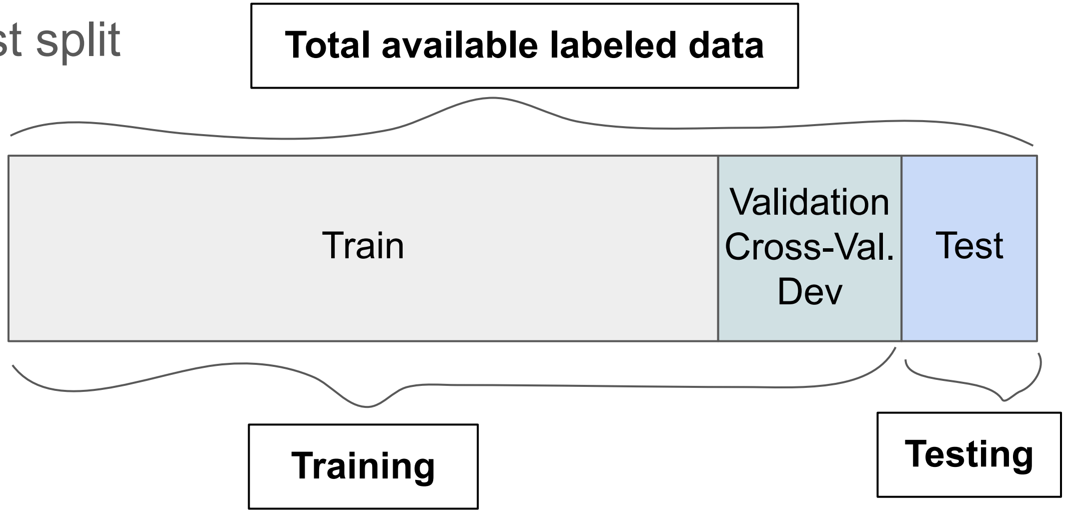
- Input data
- Model architecture
  - network of layers (parameterized by weight)
- Loss function
  - Objective function to minimize
  - discrepancy between true labels and predictions
- Optimizer
  - Determines how to update the model weights



Input Data

Dataset

- Train-val-test split



Training split of the dataset can be used for training, tuning the hyperparameters check the performance of different model architectures, etc

Testing split of the dataset should only be used when you have **finalized** your model and check the performance of your model before deploying or publication

Input Data

Dataset

```
fit(
    x=None, y=None, batch_size=None, epochs=1, verbose='auto',
    callbacks=None, validation_split=0.0, validation_data=None, shuffle=True,
    class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None,
    validation_steps=None, validation_batch_size=None, validation_freq=1,
    max_queue_size=10, workers=1, use_multiprocessing=False
)
```

Training data

Fraction of the **training data** to be used as **validation data**. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch.

Input Data

Dataset

```
fit(
    x=None, y=None, batch_size=None, epochs=1, verbose='auto',
    callbacks=None, validation_split=0.0, validation_data=None, shuffle=True,
    class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None,
    validation_steps=None, validation_batch_size=None, validation_freq=1,
    max_queue_size=10, workers=1, use_multiprocessing=False
)
```

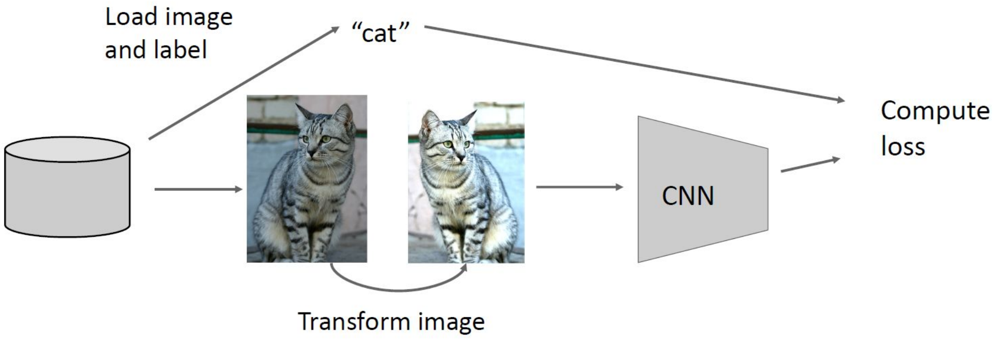
Training data

Explicitly provide **validation data** on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data.

Data Augmentation

Idea: Increase the number of training data by *randomly* **shifting/cropping/rotating** original data. Helps model generalize better.

Training: change the input data at each training step so that the CNN never sees the same image every step



Data Augmentation

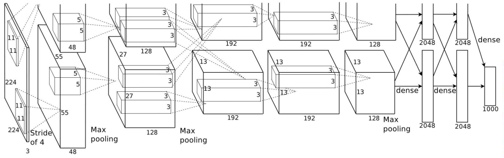
Can be incorporated in keras as a preprocessing layer in the **model**!

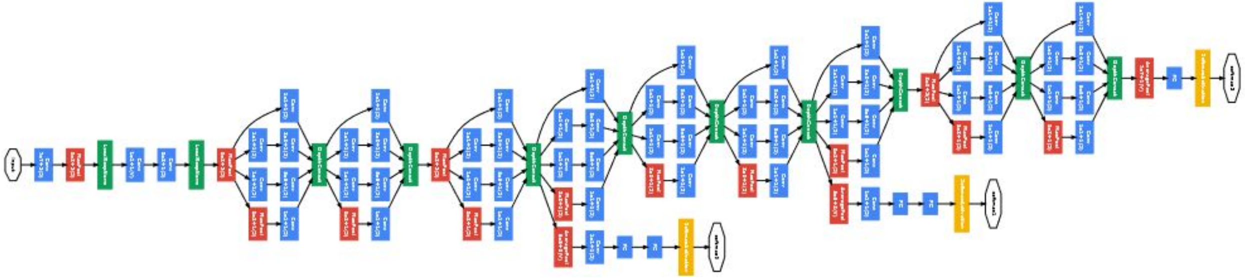
```
data_augmentation = tf.keras.Sequential([
  layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
  layers.experimental.preprocessing.RandomRotation(0.2),
])
```

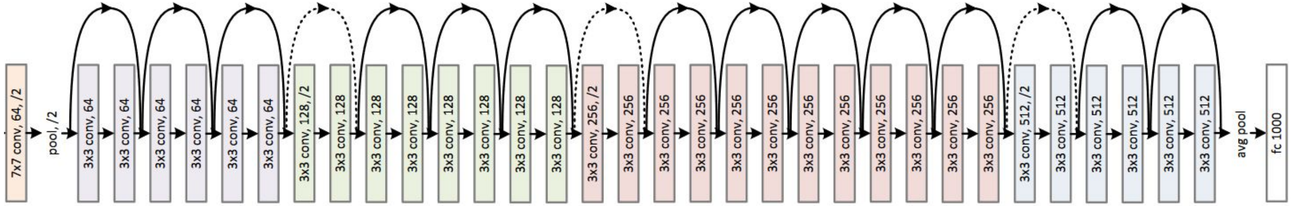
Model Architecture

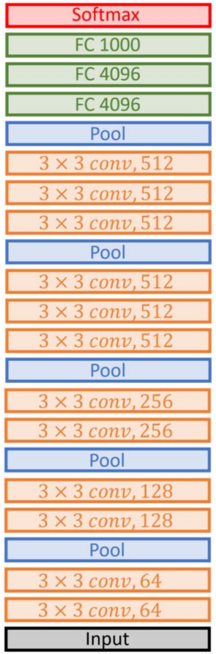
Network of layers

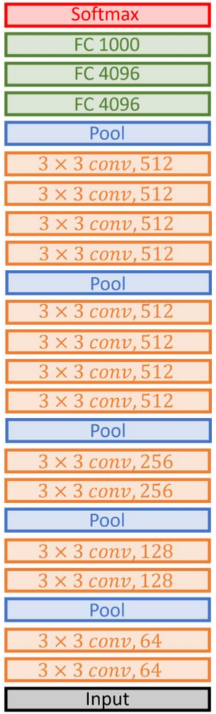
- parameterized by **weights**











Loss Function

A measure of the distance between your model's prediction and the target.

Desirable properties of the loss function are

- Predictions deviate too much from target: loss function ↑
- Predictions not too far from target: loss function ↓
- Easily differentiable (?)

Broadly classified into to categories

1. Regression loss
2. Classification loss

Loss Function

Supervised learning problem measures the compatibility between a prediction (e.g. the class scores in classification) and the ground truth label. The data loss takes the form of an average over the data losses for every individual example.

$$L = \frac{1}{N} \sum_i L_i$$

Classification Loss

1. SVM Loss

$$L_i = \sum_{j \neq y_i} \max(0, f_j - f_{y_i} + 1)$$

2. Cross Entropy Loss

$$L_{\text{cross-entropy}}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i y_i \log(\hat{y}_i)$$

Regression Loss

1. L2 Loss/MSE

$$L_i = \|f - y_i\|_2^2$$

2. L1 Loss/MAE

$$L_i = \|f - y_i\|_1$$

tf.keras.losses

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

model = keras.Sequential()
model.add(layers.Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(layers.Activation('softmax'))

loss_fn = keras.losses.SparseCategoricalCrossentropy()
model.compile(loss=loss_fn, optimizer='adam')
```

All built-in loss functions may also be passed via their string identifier:

```
# pass optimizer by name: default parameters will be used
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
```

Loss functions are typically created by instantiating a loss class (e.g. `keras.losses.SparseCategoricalCrossentropy`). All losses are also provided as function handles (e.g. `keras.losses.sparse_categorical_crossentropy`).

Using classes enables you to pass configuration arguments at instantiation time, e.g.:

```
loss_fn = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

Probabilistic losses

- BinaryCrossentropy class
- CategoricalCrossentropy class
- SparseCategoricalCrossentropy class
- Poisson class
- binary_crossentropy function
- categorical_crossentropy function
- sparse_categorical_crossentropy function
- poisson function
- KLDivergence class
- kl_divergence function

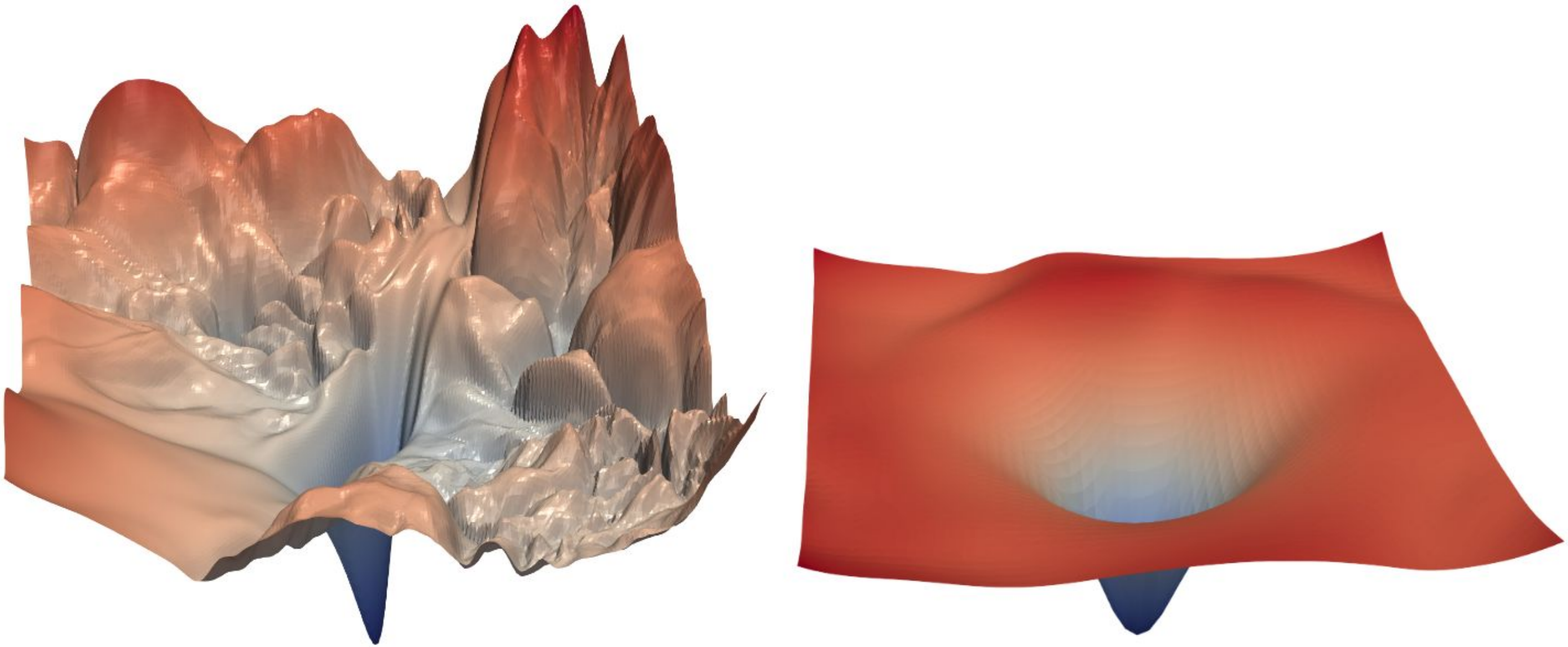
Regression losses

- MeanSquaredError class
- MeanAbsoluteError class
- MeanAbsolutePercentageError class
- MeanSquaredLogarithmicError class
- CosineSimilarity class
- mean_squared_error function
- mean_absolute_error function
- mean_absolute_percentage_error function
- mean_squared_logarithmic_error function
- cosine_similarity function
- Huber class
- huber function
- LogCosh class
- log_cosh function

Last-layer Activation & Loss

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

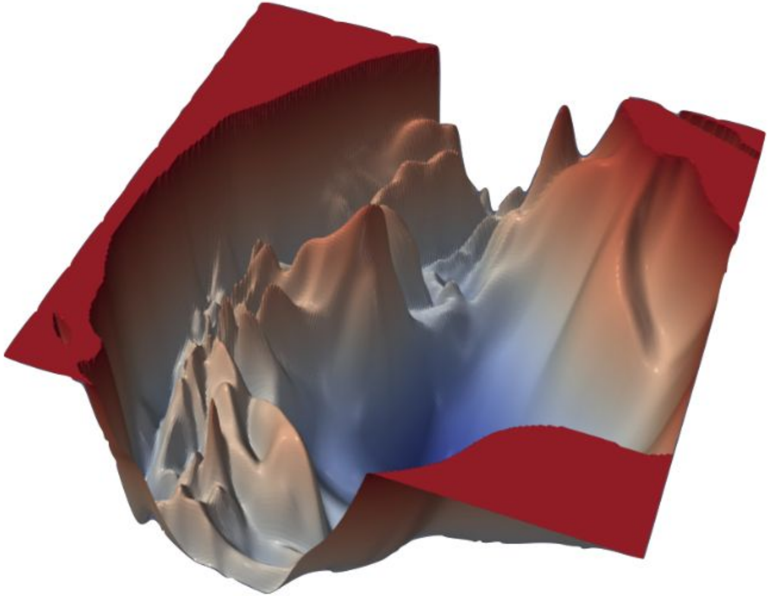
Loss surfaces of ResNet-56



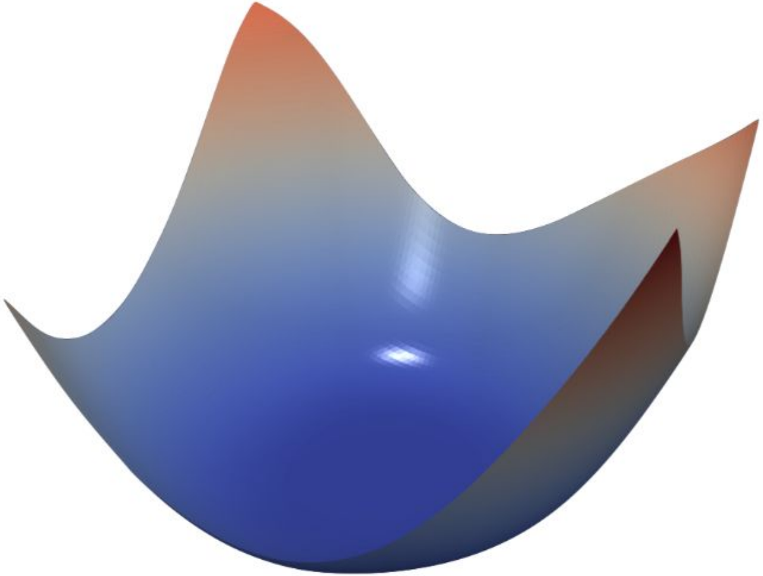
(a) without skip connections (b) with skip connections

https://arxiv.org/abs/1712.09913

Loss surfaces of ResNet-110 and DenseNet for CIFAR-10



(a) ResNet-110, no skip connections

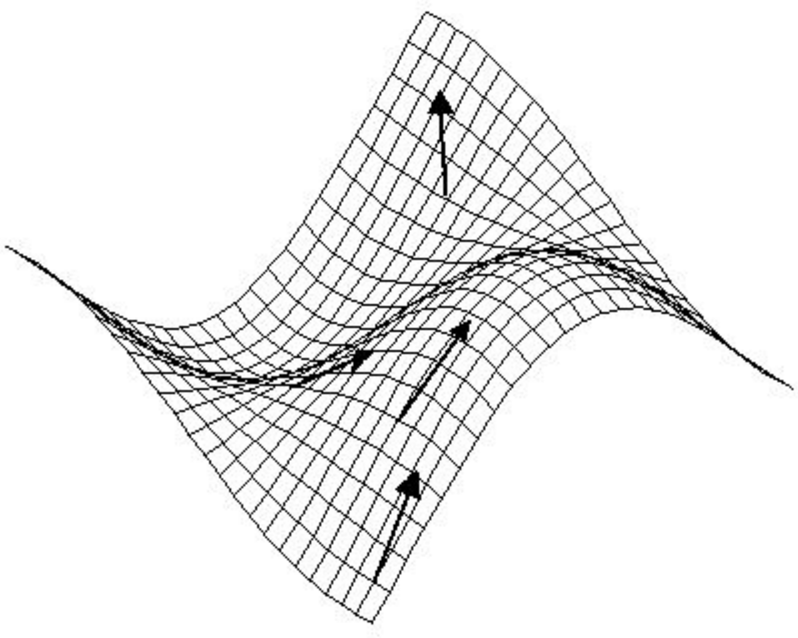


(b) DenseNet, 121 layers

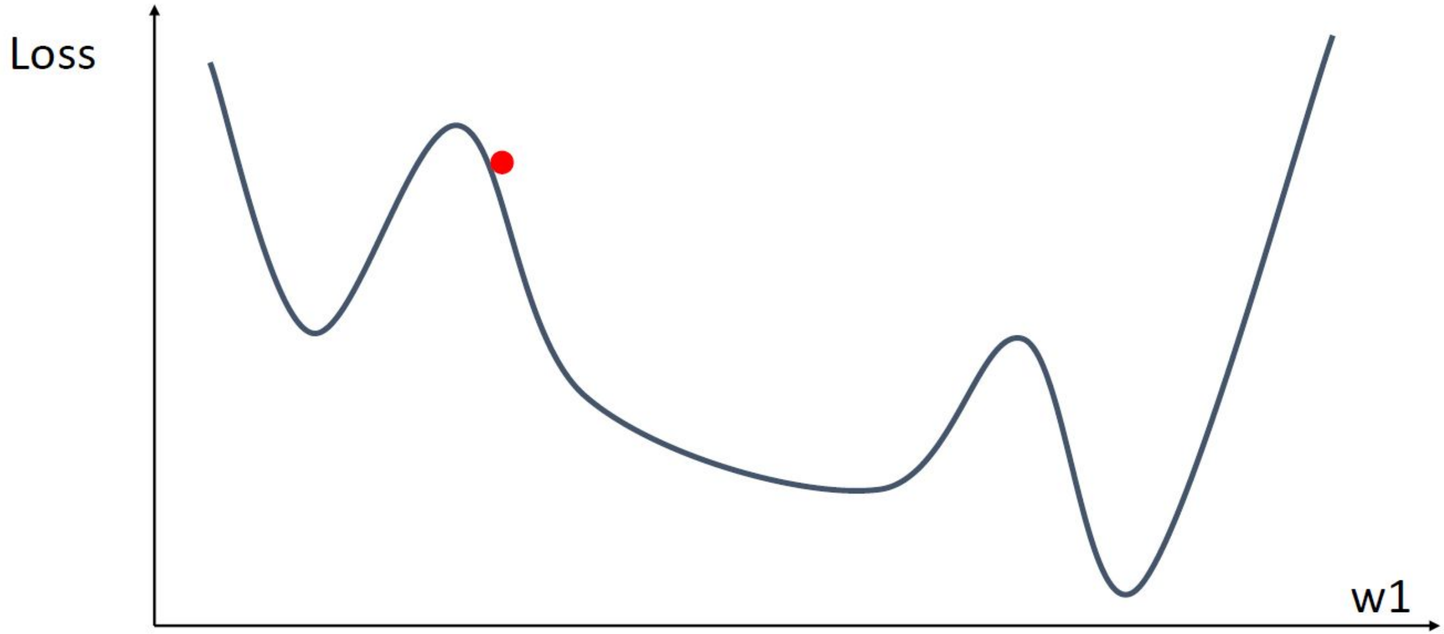
https://arxiv.org/abs/1712.09913

Gradient

The **gradient vector** can be interpreted as the "direction and rate of fastest increase"



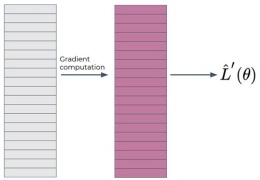
1D Example



Gradient Based Optimization: SGD

- Vanilla Gradient Descent (Batch Gradient Descent)

  $\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta L(\theta ; x^{(0:N-1)}; y^{(0:N-1)})$

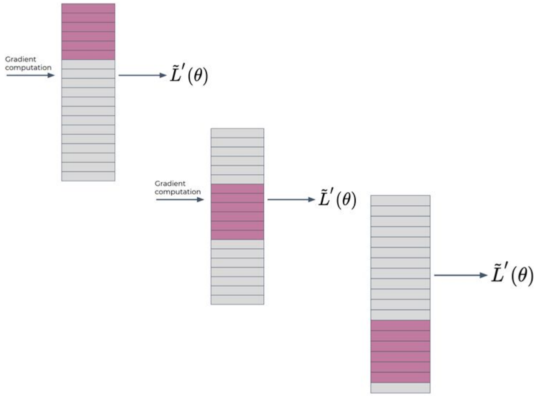


- Mini-batch Gradient Descent

  $\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta L(\theta; x^{(i:i+n)}; y^{(i:i+n)})$ [i'th mini-batch size

- Stochastic Gradient Descent

  $\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta L(\theta; x^{(i)}; y^{(i)})$



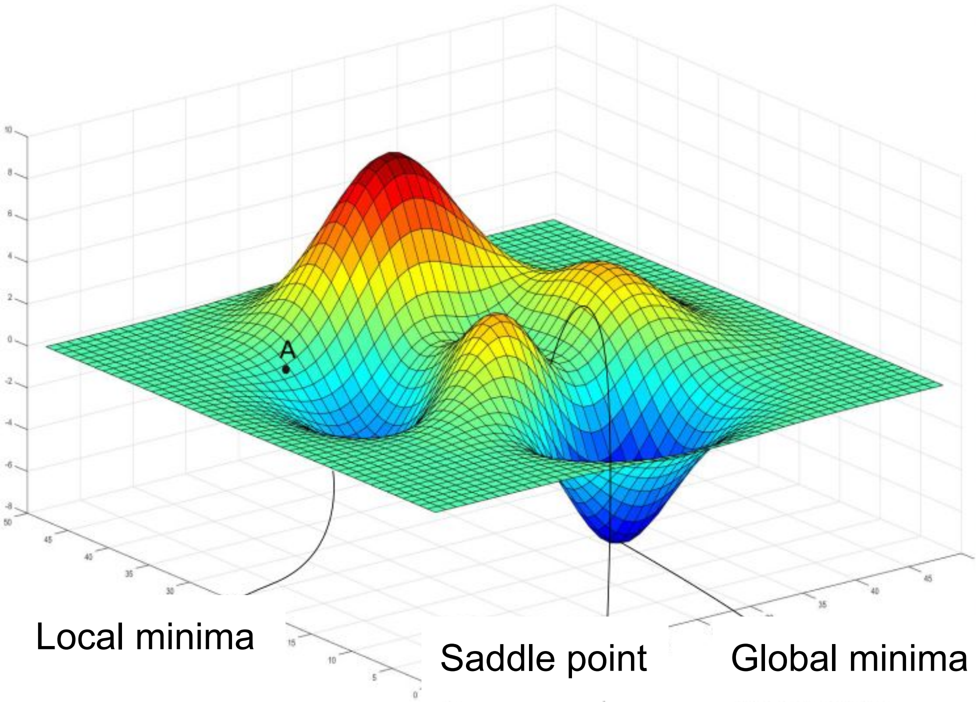
Gradient Based Optimization: SGD

Problem with SGD

- Slow update at saddle points
- Stuck at local minima

Solution

- Incorporate momentum?



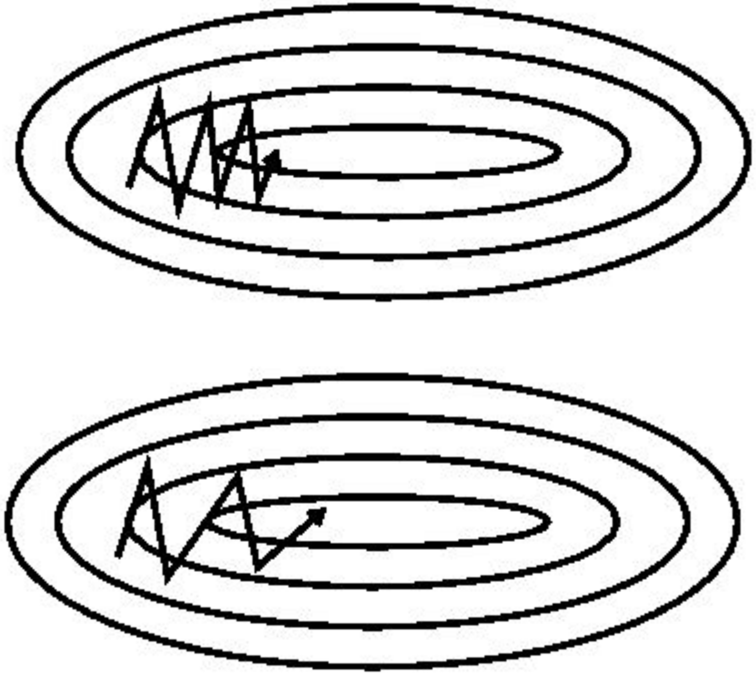
Gradient Based Optimization: SGD+Momentum

- Gradient Descent with Momentum

  Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations

  $v_t = \gamma \cdot v_{t-1} + \eta \cdot \nabla_\theta L(\theta)$

  $\theta_{t+1} = \theta_t - v_t$



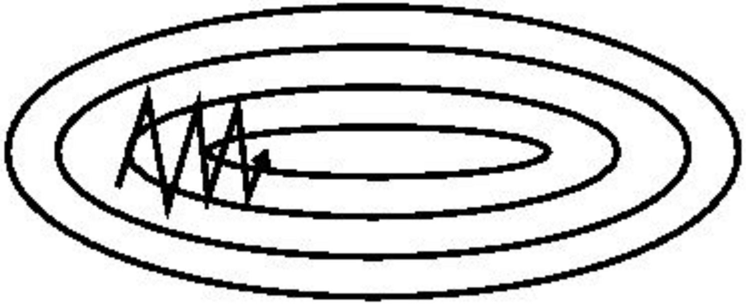
Gradient Based Optimization: Adaptive Gradient

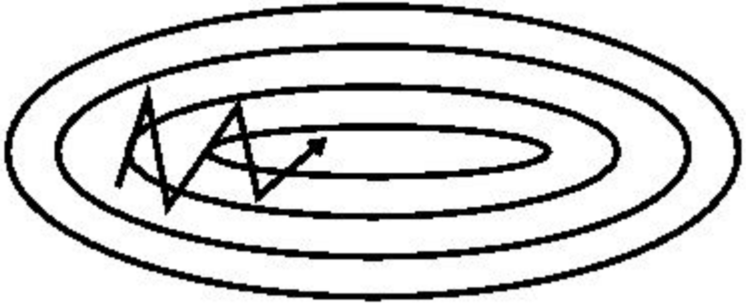
- AdaGrad

  increases the learning rate for sparser parameters and decreases the learning rate for ones that are less sparse

  $G_t = G_{t-1} + (\nabla_\theta L(\theta))^2$

  $\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta L(\theta) / \sqrt{(G_t + \epsilon)}$





Gradient Based Optimization: RMSProp

- RMSProp

  AdaGrad but with exponential averaging the square of the gradient

  $G_t = \gamma \cdot G_{t-1} + (1 - \gamma) \cdot (\nabla_\theta L(\theta))^2$

  $\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta L(\theta) / \sqrt{(G_t + \epsilon)}$

Gradient Based Optimization: Adaptive Moment

- Adam (Simplified)

  Takes the idea of RMSProp + Momentum

  $v_t = \gamma_1 \cdot v_{t-1} + (1 - \gamma_1) \cdot \nabla_\theta L(\theta)$

  $G_t = \gamma_2 \cdot G_{t-1} + (1 - \gamma_2) \cdot (\nabla_\theta L(\theta))^2$

  $\theta_{t+1} = \theta_t - \eta \cdot v_t \;/\; \sqrt{(G_t + \epsilon)}$

tf.keras.optimizers

Classes

`class Adadelta` : Optimizer that implements the Adadelta algorithm.

`class Adagrad` : Optimizer that implements the Adagrad algorithm.

`class Adam` : Optimizer that implements the Adam algorithm.

`class Adamax` : Optimizer that implements the Adamax algorithm.

`class Ftrl` : Optimizer that implements the FTRL algorithm.

`class Nadam` : Optimizer that implements the NAdam algorithm.

`class Optimizer` : Base class for Keras optimizers.

`class RMSprop` : Optimizer that implements the RMSprop algorithm.

`class SGD` : Gradient descent (with momentum) optimizer.

tf.keras.optimizers Examples

Optimizer parameters

```
tf.keras.optimizers.Adam(
    learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07, amsgrad=False,
    name='Adam', **kwargs
)
```

```
tf.keras.optimizers.RMSprop(
    learning_rate=0.001, rho=0.9, momentum=0.0, epsilon=1e-07, centered=False,
    name='RMSprop', **kwargs
)
```

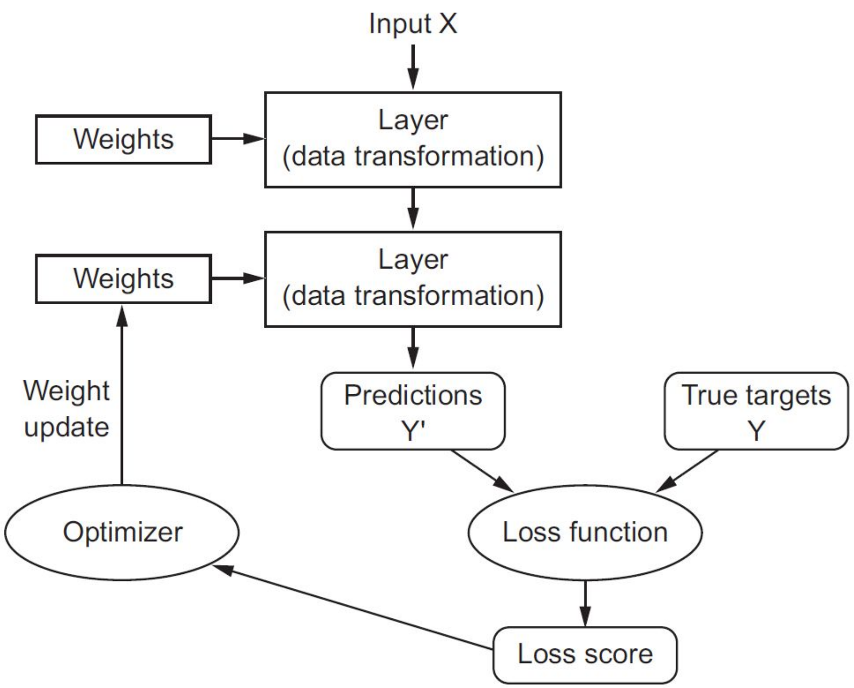
Model Compilation in Karas

```
compile(
    optimizer='rmsprop', loss=None, metrics=None, loss_weights=None,
    weighted_metrics=None, run_eagerly=None, steps_per_execution=None, **kwargs
)
```

Deep Learning Pipeline

Deep Learning Pipeline

- Input data
- Model architecture
  - network of layers
- Loss function
  - Objective function to minimize
  - discrepancy between true labels and predictions
- Optimizer
  - Determines how to update the model using some variant of gradient descent



Transfer Learning

For a particular trak, training a CNN from scratch can be challenging

- Not enough data
- Computational resources

Transfer learning allows you overcome this problem by using pre-trained CNNs

- Use pre-trained CNNs as feature extractors
- Works well even for small datasets

Transfer Learning

A pretrained network is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task (ImageNet)

If this original dataset is large enough and general enough, then the spatial hierarchy of features learned by the pretrained network can effectively act as a generic model of the visual world

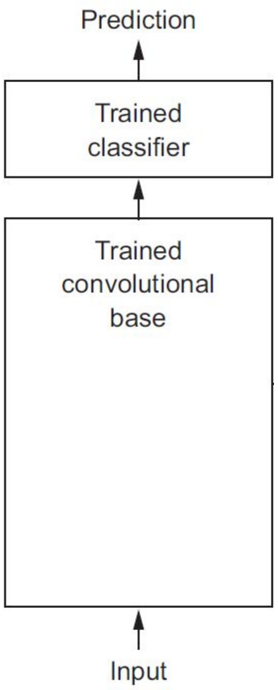
Its features can prove useful for many different computer vision problems, even though these new problems may involve completely different classes than those of the original task.

Transfer Learning

A pretrained network

Trained CNN base + Trained Dense classifier

a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task (ImageNet)

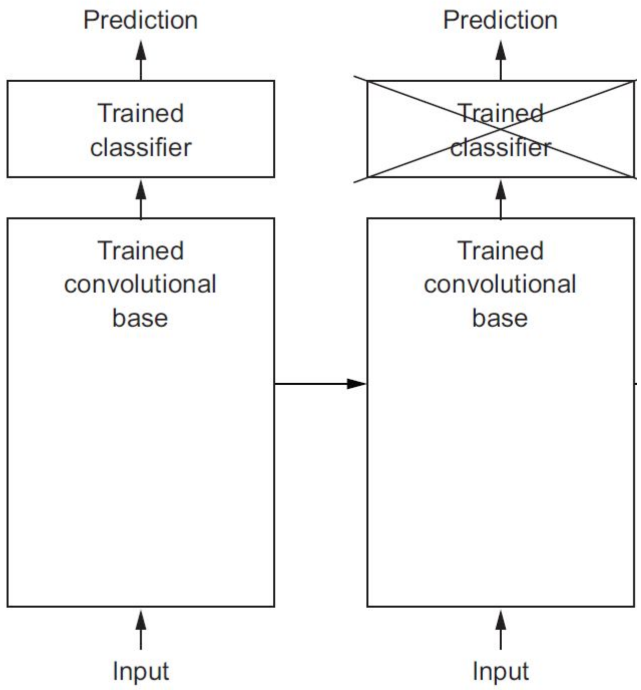


Transfer Learning

A pretrained network

Trained CNN base + ~~Trained Dense classifier~~

a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task (ImageNet)

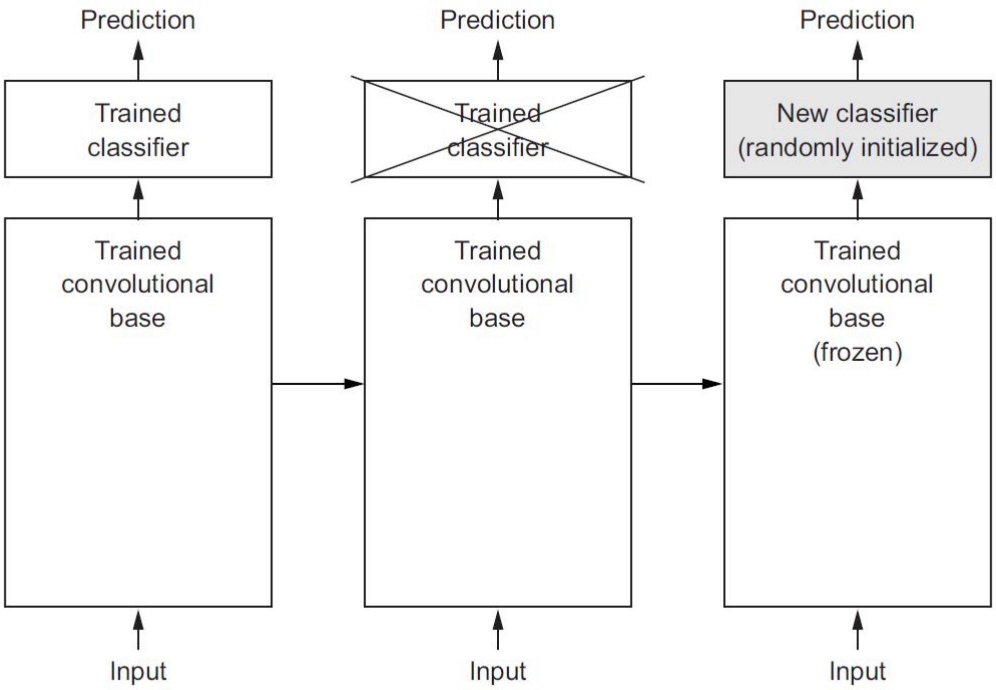


Transfer Learning

A pretrained network

Trained CNN base + ~~Trained Dense classifier~~ + New classifier

Train the new classifier on your own data

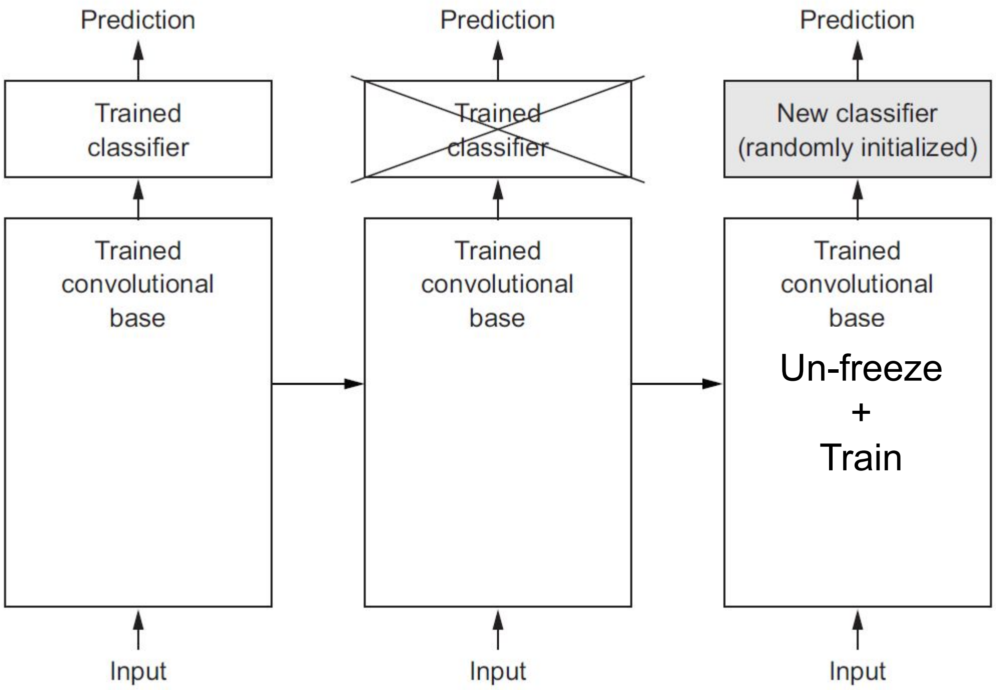


Transfer Learning

Fine tuning

Trained CNN base + New classifier

Unfreeze the base and fine-tune the new classifier on your own data with very small learning rate



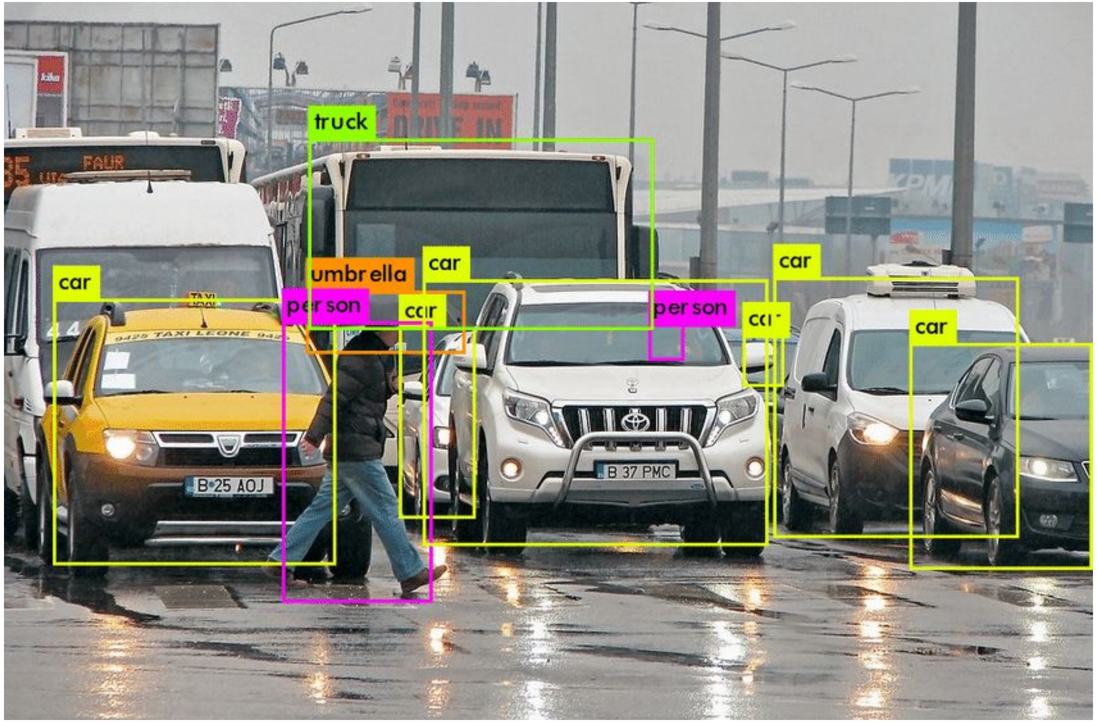
tf.keras.applications

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time per inference step (CPU)	Time per inference step (GPU)
Xception	88 MB	0.790	0.945	22,910,480	126	109.42ms	8.06ms
VGG16	528 MB	0.713	0.901	138,357,544	23	69.50ms	4.16ms
VGG19	549 MB	0.713	0.900	143,667,240	26	84.75ms	4.38ms
ResNet50	98 MB	0.749	0.921	25,636,712	-	58.20ms	4.55ms
ResNet101	171 MB	0.764	0.928	44,707,176	-	89.59ms	5.19ms

+ many more

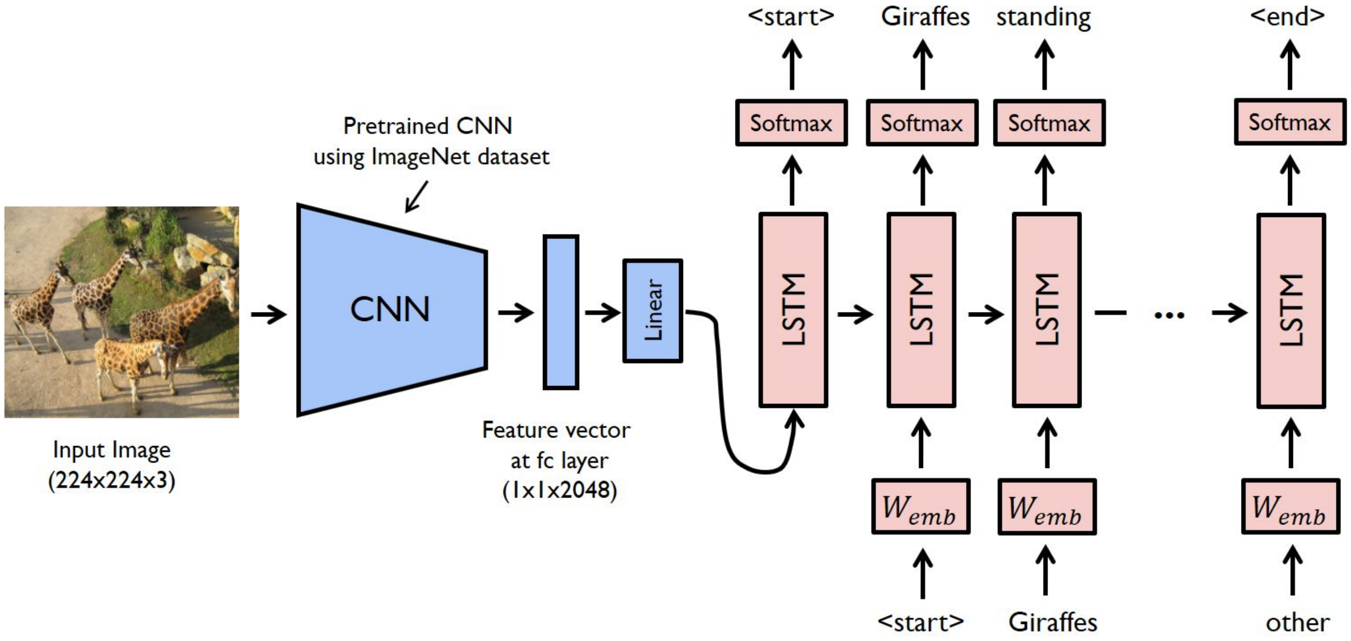
https://keras.io/api/applications/

Application: Object Detection

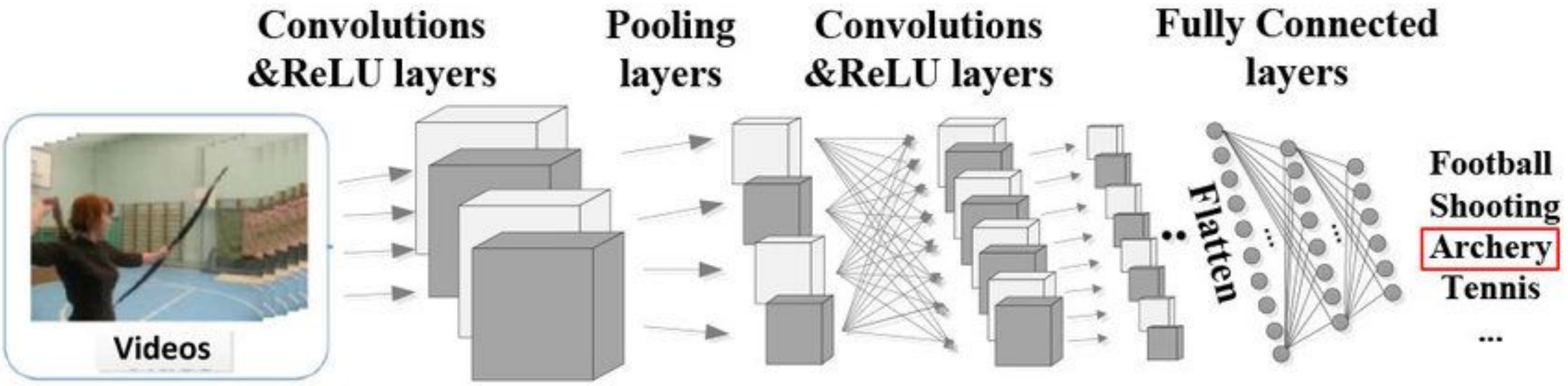


Application: Image Segmentation

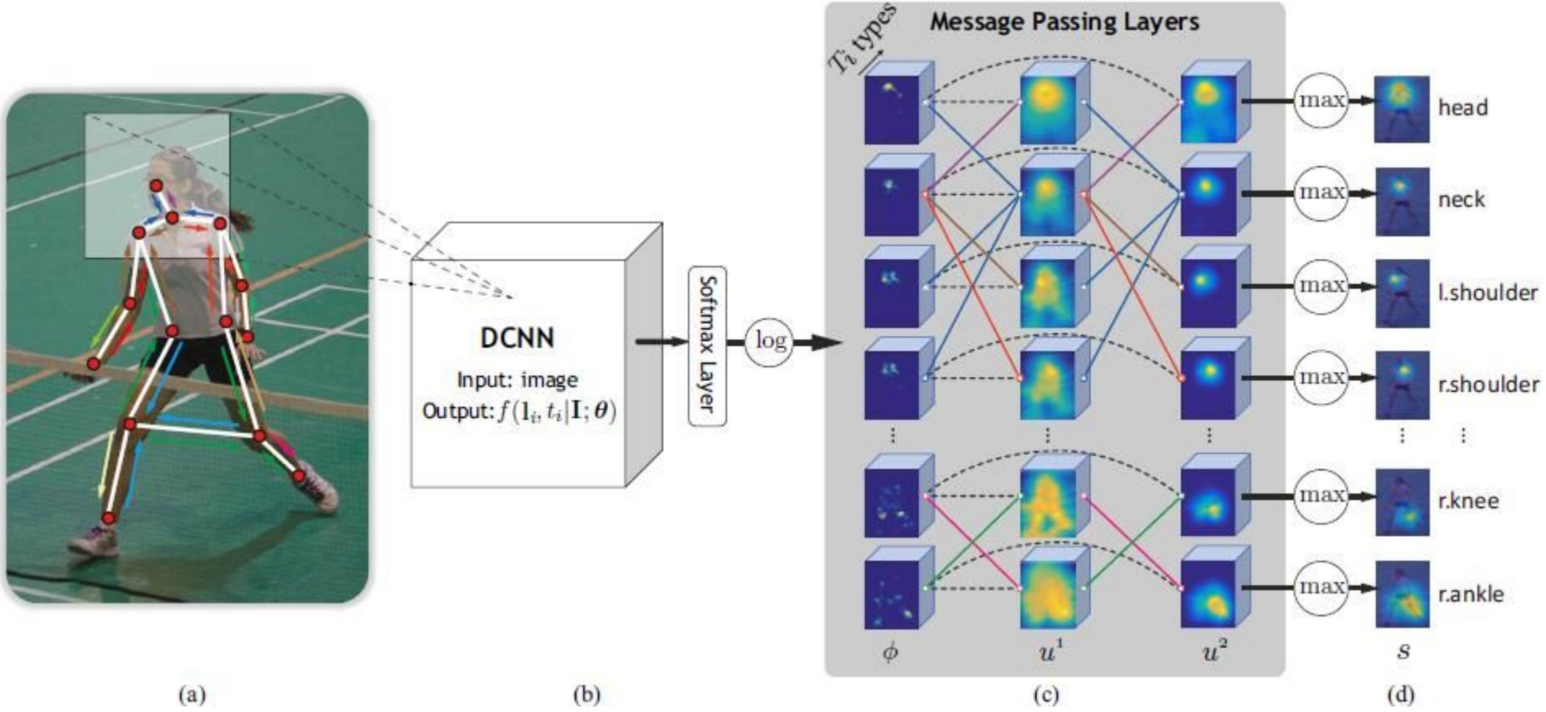
Application: Caption Generation



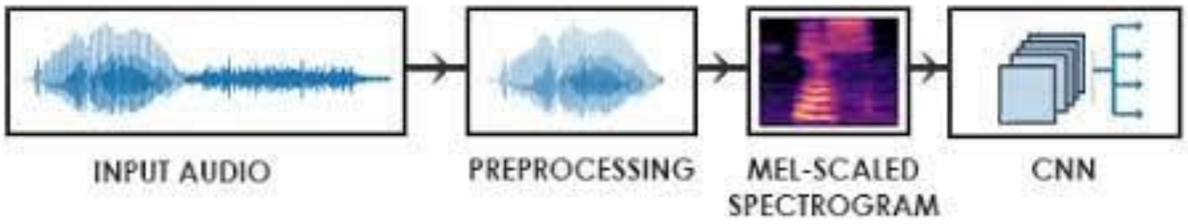
Application: Video Classification



Application: Human Pose Estimation



Application: Sound Classification



Resources

1. https://cs231n.github.io/convolutional-networks/
2. https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2020/
3. https://www.tensorflow.org/api_docs/python/tf/keras
4. Deep Learning with Python Book by François Chollet
5. https://www.deeplearningbook.org/
6. Hands-on Computer Vision with TensorFlow 2 by Eliot Andres & Benjamin Planche (Packt Pub.)
7. Deep Learning with PyTorch Book by Eli Stevens and Thomas Viehmann