Deep Learning Notes

Mauricio Barba da Costa

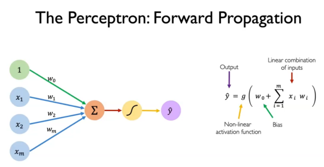
February 16, 2021

1 Monday, January 18

These neural network algorithms were developed a long time ago but they've become a lot more prominent because of big data, better hardware, and better software.

1.1 The Perceptron

The structural building block of the neural network


The Perceptron: Forward Propagation
1
x_1
x_2
x_m
w_0
w_1
w_2
w_m
Σ
$\hat{y}$
Output
Linear combination of inputs
$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$
Non-linear activation function
Bias

We can express this as

$$\hat{y} = g(w_0 + X^T W)$$

where

$$X = \begin{pmatrix} x_1 \\ \dots \\ x_m \end{pmatrix} \quad \text{and} \quad W = \begin{pmatrix} w_1 \\ \dots \\ w_m \end{pmatrix}$$

w_0 is called a **bias term**, g is a **nonlinear activation function**. One example of such a function is the **sigmoid function**.

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

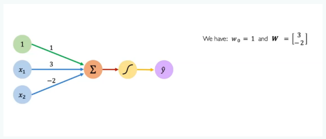
This is obtained in the TensorFlow library by `tf.math.sigmoid(z)`. The image of the sigmoid function is always between 0 and 1 and its domain is the real numbers.

Why do we need activation functions?

To introduce nonlinearities in our network. We need to deal with nonlinear data. Introducing nonlinearities allow us to approximate arbitrarily complex functions.

Example 1

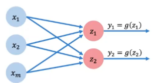
Suppose we have two inputs x_1 and x_2.


1
x_1
x_2
1
3
-2
Σ
$\hat{y}$
We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

The perceptron has 3 steps:

1. Take the dot product of the inputs and the weights
2. Add a weight bias
3. Apply a nonlinearity

If we want to add more outputs, we can add another perceptron.



$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

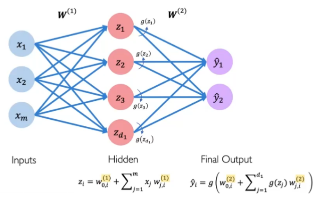
Let's make a dense layer from scratch

```
class MyDenseLayer(tf.keras.layers.Layer):
    def __init__(self, input_dim, output_dim):
        super(MyDenseLayer, self).__init__()

        # Initialize weights and bias
        self.W = self.add_weight([input_dim, output_dim])
        self.b = self.add_weight([1, output_dim])

    def call(self, inputs):
        # Forward propagate the inputs
        z = tf.matmul(inputs, self.W) + self.b

        # Feed through a non-linear activation
        ouput = tf.math.sigmoid(z)
```

We don't have to do all that stuff since tensor flow has it preset for us

```
import tensorflow as tf
layer = tf.keraslayers.Dense(units=2)
```

1.2 A Single Layer Neural Network



We can create this exact neural network (complete, two layers) like this:

```
import tensorflow as tf
model = tf.keras.Sequential([
    tf.keras.layers.Dense(n),
    tf.keras.layers.Dense(2)
])
```

Observe that the first column is reserved for the inputs ($x_1, \ldots, x_m$). All we have to do is stack these layers to get more hierarchical models.

```
import tensorflow as tf
model = tf.keras.Sequential([
    tf.keras.layers.Dense(n_1)
    tf.keras.layers.Dense(n_2)
    ...
    tf.keras.layers.Dense(2)
])
```

Example 2 (Will I pass this class?)
x_2 is the hours spent on the final project.

x_1 is the number of lectures you attended.

$\hat{y}$ is a number between 0 and 1 that's supposed to give us the likelihood I pass the class. We're going a employ another useful tool for neural networks

$$\mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

The **loss** of a network measures the cost incurred from incorrect predictions. The first input is the predicted and the second in the actual. Close implies small loss and far implies large loss. We want to minimize the loss

The **empirical loss** measures the total loss over our entire dataset. It is calculated by

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

This is called the objective function, cost function, empirical risk. We seek to minimize $J(W)$. A **cross entropy** loss can be used to models that output a probability between 0 and 1.

$$J(W) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)}\log(f(x^{(i)}; W)) + (1 - y^{(i)}) log(1 - f(x^{(i)}; W))$$

The y and $1 - y$ components represet the actual and the f represents the predicted.

```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, predicted)
```

The **Mean Squared Error Loss** can be used with regression models that output continuous real numbers.

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - f(x^{(i)}; W))^2$$

```
loss = tf.reduce_mean(tf.square(tf.substract(y, predicted)))
loss = tf.keras.losses.MSE(y, predicted)
```

1.3 Training the Neural Network

We seek to find the network weights taht achieve the lowest loss

$$W^* = \text{argmin}_W \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

$$W^* = \text{argmin}_W J(W)$$

We can compute the gradient,

$$\frac{\partial J(W)}{\partial W}$$

This gives us the direction of steepest ascent. What we'll do is move a small step in the direction of the negative of the gradient. Then calculate the gradient again and perform the same process. This process is called gradient descent. The algorithm is as follows:

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. (a) Compute gradient $\frac{\partial J(W)}{\partial W}$
   (b) Update weights, $W = W - \eta \frac{\partial J(W)}{\partial W}$
4. Return weights

```
imoprt tensorflow as tf
weights = tf.Variable([tf.random.normal()])
while True:
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

        weights = weights - lr * gradient
```

1.4 Backpropagation

$$\frac{\partial J(W)}{w_1} = \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} \frac{\partial z_1}{\partial w_1}$$

Do this for every weight in the neural network. The η that was used in the algorithm description is called the learning rate. "How much do we want to trust that gradient". How do we furnish stable learning rates?

One idea might be to try a bunch of learning rates.

How about we make a learning rate that fixes itself.

There are a bunch of optimizers available.

1.5 Putting it all together

```
import tensorflow as tf
model = tf.keras.Sequential([...])

# pick your favorite optimizer
optimizer = tf. keras.optimizer.SGD()
```

```
while True:

    prediction = model(x)

    with tf.GradientTape() as tape:
        loss = compute_loss(y, prediction)
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainabile))
```

Gradient descent can be very computational intensive.

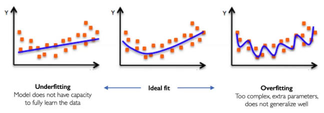
Let's set

$$\frac{\partial J(W)}{\partial W} = \frac{1}{B}\sum_{k=1}^{B}\frac{\partial J_k(W)}{\partial W}$$

This allows us to estimate the gradient descent of several points having chosen only B data points. This allows massive parallelization.

1.6 Overfitting

This is the hardest problem in neural networks.



How do we make sure our models don't overfit. We want to improve the generalization of our networks. When we say there are n layers in an NN, that means that there are two hidden networks.

During training, randomly set some activations to 0. Typically this is 50% of activations in layer. It forces to not rely on any 1 node.

`tf.keras.layers.Dropout(p=0.5)` Forces the network to find several pathways so it doesn't depend on any 1 pathway too much. This makes it more "generalizable". This is one type of **regularization** technique (trying to prevent overfitting)

2 Tuesday, January 19

Applying neural networks to to problems that involve sequential modelling of data. We need a different architecture from what we saw last time. **Sequential modelling and processing**: Given an image of a ball, where will it go next? What if we also saw the previous positions of the ball? Sequential data has, well, a sequence. In the previous example with a student passing the class, there was no notion of sequence.

We can classify some types of sequence modeling applications:

- One to one; e.g. binary classification.
- Many to one; e.g. sentiment classificaion. at the end, our task is to predict a single output, the sentiment. The input is the sequence of words
- One to many; e.g. image captioning. A single input then the output is a sequence of things
- Many to many; e.g. machine translation. Given a sequence of words in one language, the output is a sequence of words in another.

2.1 Neurons with Recurrence

How do we add the temporal dimension into our models? We need to link computations of the network at different timesteps to eachother. We introduce this notion of a cell state. It can be passed from timestep to timestep. By having this recurrence relation, we capture some notion of memory.

$$\hat{y}_t = f(x_t, h_{t-1})$$

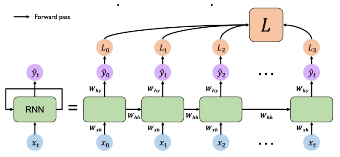
h denotes past cell states.

2.2 Recurrent Neural Networks (RNNs)

RNNs have a state h_t that is updated at each time step as a sequence is processed.

```
my_rnn = RNN()
hidden_state = [0,0,0,0]
sentence = ['I', 'love', 'recurrent', 'neural']
for word in sentence:
    prediction, hidden_state = my_rnn(word, hidden_state)
next_word_prediction = prediction
```

RNN computations include both the internal cell update and the output prediction.



```
class MyRNNCell(tf.keras.layers.Layer):
    def __init__(self, rnn_units, unput_dim, output_dim):
        super(MYRNNCell, self).__init__()

        self.W_xh = self.add_weight([rnn_units, input])
        self.W_hh = self.add_weight([rnn_units, rnn_units])
```

```
            self.W_hy = self.add_weight([output_dim, rnn_units])

            self.h = tf.zeros([rnn_units, 1])

        def call(self, x):
            self.h = tf.math.tanh(self.W_hh * self.h + self.W_xh * x)

            output = self.W_hy * self.h

            return output, self.h


    tf.keras.layers.SimpleRNN(rnn_units)
```

Why are RNNs good for handling sequential data? Because they take into account some notion of 'memory'.

We need to convert words into a vector representation. **embedding** transforms indexes into a vector of fixed size. It captures the content of the input. Here's how we might do it.

1. Furnish a bijection between a set of words and natural numbers.
2. Then given a natural number i, output e_i.

We want words that are similar to each other to be closer to each other.

2.3 Backpropagation Through Time (BPTT)

This is how we train an RNN.

Computing the gradient with respect to h_0 involves many factors of $W_h h$ and repeated gradient computation. If we have many values >1, we get **exploding gradients** and if we get many values <1 we get vanishing gradients.

Use a more complex recurrent unit with gates to control what information is passed through

LSTM networks rely on a gated cell to track information throughought many time steps.

2.4 Long Short Term Memory (LSTM) Networks

How do LSTMs work?

```
    tf.keras.layers.LSTM(num_units)
```

Information is added or removed through structures called **gates**. Gates optionally let information through, for example via a sigmoid neural net layer. LSTM do the following:

1. Forget old irrelevant memory
2. Store new important memory

3. Update internal cell state
4. Output

Forgetting is achieved through passing one of the previous states through one of the sigmoid gates. We next want to determine what part of the old information is relevant. LSTMs have both a c_t and an h_t. The c_t is selectively updates by passing the h_t through the sigmoid gates.

The output gate controls what information is sent to the next time step. The key takeaway is that LSTMs regulate information flow and storage. They can capture longer term dependencies.

Example 3 (Music generation)
We're creating an NN that takes in musical notes and then we want to predict what the next couple of notes are.

Another example might be take a sentence and gauge the sentiment of the sentence.

```
loss - tf.nn.softmax_cross_entropy_with_logits(y, predicted)
```

There is an encoding bottleneck. It's hard to parallelize and it's slow.

3 Wednesday, January 20

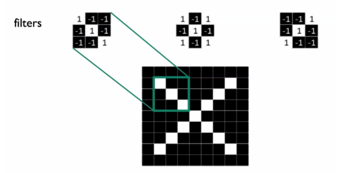
3.1 What Computers See

How does a computer process an image? Images are just images to a computer. 2 dimensional lists of numbers to be exact. A pixel is just a number. A color is represented by 3 numbers in a pixel. This is how a computer sees an image. **Regresssion**: ouput variable takes continuous values **Classification**: output variable takes class label. Can produce probability of belonging to a particular class.

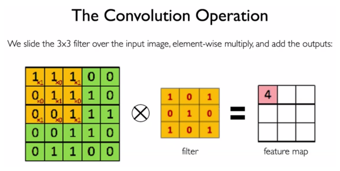
Classification is done by detecting the features of an image. For example, a nose, eyes, a mouth.

3.2 Feature extraction and convolution

We want to be able to classify an X as an X even if it's shrunk, or undergoes some natural transformation. Our model should identify important features of X to be able to idenfity the letter.



The smaller matrices of filters represent the filters of weights in order to detect the features in the input image.


The Convolution Operation
We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:
filter
feature map

By changing the weights of the filters of a convolution, we can learn to detect different features of an image.

3.3 Convolutional Neural Networks

In TensorFlow language:

```
tf.keras.layers.Conv2D
tf.keras.activations.*
tf.keras.layers.MaxPool2D
```

In real life:

1. Convolution. Applly filters to generate feature maps
2. Nonlinearity. Often ReLu
3. Pooling. Downsampling operation on each feature map

Train the model with image data. Learn weights of filters in convolutional layers.

There are a lot of things we can think about

- Layer dimensions
- Stride. Filter step size
- Receptive Field. Locations in input image that a node is path connected to

```
tf.keras.layers.Conv2D(filters=d,kernel_size=(h,w),strides=n)
```

Apply ReLU after every convolution operation. After convolutional layers. ReLU is rectified linear unit. It's zero when the input is negative and the identity elsewhere.

```
tf.keras.layers.MaxPool2D(
    pool_size=(2,2),
    strides=2
)
```

CONV and POOL layers output high-level features of input Fully connected layer uses these features for classifying input image Express output as probability of image belonging to a particular class.

```
import tensorflow as tf
```

```
def generate_model():
    model = tf.keras.Sequential([
        # First convolutional layer
        tf.keras.layers.Conv2D(32, filters_size=3, activation='relu')
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),

        # second convolutional layer
        tf.keras.layers.Conv2D(64, filter_size=3, activation='relu'),
        tf.layers.MaxPool2D(pool_size=2, strides=2),

        # fully connected classifier
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(1024, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')
    ])

    return model
```

4 Thursday, January 21

Why are generative models important? Debiasing. They are capable of uncovering underlying features in a dataset. Datasets can sometimes be consistent in a way that isn't representative of real life.

How can we detect when we encounter something new or rare? Leverage generative models, detect outliers in the distribution. Use outliers during training to improve even more.

4.1 Latent Variable Models

What is a latent variable?

{ Insert Plato's parable of the cave }.

Variables are not exactly observable but are the explanatory factors.

4.2 Autoencoders (subtype of latent variable models)

Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data. We might want to train a model to use features to reconstruct the original data.

4.3 Variational autoencoders

Traditional goes from input to reconstructed output. Variational autoencoders impose a stochastic twist to generate smoother representations of the data. We get the mean and variation. The encoder computes $q_\phi(z|x)$ and the decoder computes $p_\theta(x|z)$.

$\mathcal{L}(\phi, \theta, x)$ is the reconstruction loss plus the regularization term. The loss terms are reprsented by ϕ, θ. q is a probability distrubution on z with a prior on x.

$$D(q_\phi(z|x)||p(z))$$

By imposing the regularization term, we can prevent the network from overfitting.

What might be a good choice of prior, $p(z)$? A common choice of prior is a normal Gaussian distribution $p(z) = \mathcal{N}(\mu = 0, \sigma^2 = 1)$.

- This encourages encodings to distribute encodings evenly around the center of the latent space
- Penalize the network when it tries to "cheat" by clustering points in specific regions (i.e. by memorizing data)

$$D(q_\phi(z|x)||p(z)) = -\frac{1}{2}\sum_{j=0}^{k-1}(\sigma_j + \mu_j^2 - 1 - \log \sigma_j)$$

Why do we want to regularize and why do we select a normal prior?

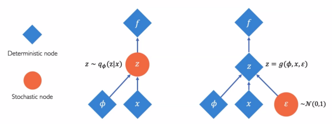
Continuity points that are close in latent space have similar content after decoding

Completeness sampling from latent space gives "meaningful" content after decoding.

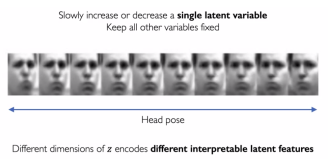
Without regularization there could be instances of points that are similar in latent space but aren't decoded similarly. The more we regularize, the greater risk we have of losing critical information.

We can't backpropagate now because of this stochasticity element. Backpropagation requires determinism.

The key is to consider the sampled latent vector z as a sum of a fixed μvector, and fixed σ vector, scaled by random constants drawn from the prior distribution. Thus, $z = \mu + \sigma \oplus \epsilon$.



We can slowly increase or decrease a single latent variable while keeping all other variables fixed.


Slowly increase or decrease a **single latent variable**
Keep all other variables fixed
Head pose
Different dimensions of z encodes **different interpretable latent features**

Ideally, we want latent variables that are uncorrelated with each other. Enforce diagonal prior on the latent variables to encourage independence. One way to disentangle:

$$\mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}, \beta) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(\mathbf{x}|\mathbf{z})] - \beta D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))$$

Latent variables are helpful for debiasing models. There's no need for us to prescribe which features are important for debiasing.

- Compress representation of world to something we can use to learn
- Reconstruction allows for unsupervised learning
- Reparameterization trick to train end-to-end
- Interpret hidden latent variables using perturbation
- Generating new examples

4.4 Generative Adversarial Network

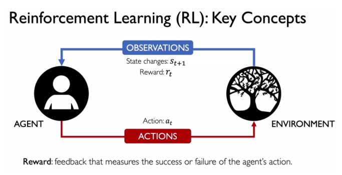
The idea is that we don't explicitly model density, and instead just sample to generate new instances.

The problem is that we want to sample from complex distribution –can't do this directly.

The solution is that we sample from something simple like noise, learn a transformation to the data distribution. **GANs** are a way to make a generative model by having two neural networks compete with each other. The **discriminator** tries to identify real data from fakes created by the generator. Knowing this, the **generator** tries to make even better images to beat the discriminator.

We want to maximize the probability that fake images are identified as fake.

5 Friday, January 22


Reinforcement Learning (RL): Key Concepts
OBSERVATIONS
State changes: s_{t+1}
Reward: r_t
AGENT
ENVIRONMENT
Action: a_t
ACTIONS
Reward: feedback that measures the success or failure of the agent's action.

$$A(s_t, a_t) = \mathbb{E}[R_t|s_t, a_t]$$

Can we determine, given the state we're in , what is the best action to take? Ultimate, the agent needs a **policy** $\pi(s)$ to infer the best action to take at its state, s.

The strategy is to choose the policy that maximizes future reward.

$$\pi^*(s) = \text{argmax}_a Q(s, a)$$

Deep Reinforcement learning algorithms are categorized under value learning and policy learning.

- Value learning is based on finding $Q(s, a)$ and $a = \text{argmax}_a Q(s, a)$.

- Policy learning is based on finding $\pi(s)$ by sampling $a \sim \pi(s)$.

Q functions are hard for humans to define. Deep Q networks. How can we use deep neural networks to model Q-functions? We can get a Q loss function too.

$$\mathcal{L} = \mathbb{E}[||(r + \gamma \max_{a'} Q(s', a')) - Q(s, a)||^2]$$

In order for the NN to find the optimal policy, it needs to look at all Q values and maximize it. It was discovered that DQN could be used to solve several Atari games.

Complexity

- Cannot handle continuous action spaces
- Can model scenarios where the action space is discrete and small

Flexibility

- Policy is deteministically computed from the Q function by maximizing the
- reward so they cannot learn stochastic policies.

In value learning, we try to have a neural network learn the Q value. It then tries to determine the best action to take given the current state. Policy learning tries to directly optimize the policy $\pi(s)$. A policy gradient allows modeling of continuous action space.

1. initialize the agent
2. run a policy until termination
3. record all states, actions, rewards
4. decrease probability of actions that resulted in low reward
5. increase probability of actions that resulted in high reward

Reinforcement learning is used for self-driving cars, the game of go.
Foundations

- Agents acting in environments
- state-action pairs $\implies$ maximize future reward
- Discounting

Q-Learning

- This is based on finding a Q-function:expected total reward given s, a.
- Policy determined by selecting action that maximizes Q function.

Policy Gradients

- Learn and optimize the policy directly
- Applicable to continuous action spaces

6 Monday, January 25

Neural Networks are excellent function approximators. Interpolating is easy but extrapolating is hard. What happens in the out of distribution data. How do we know when our neural network is not confident in the prediction it's making. Deep learning are not an all-encompassing solution that can be applied to any problem. The model is going to be as good as the training data. An autopilot in a Tesla crashed into a barrier one time because of new construction.

- **aleatoric uncertainty**
- We need uncertainty metrics to access the network's confidence in its prediction. This is **epistemic uncertainty**
- Take an input image. A temple for example, apply some perturbation to the image to get an adversarial example. The compute will then classify the temple as an ostrich. Adversarial examples don't even have to be digitally doctored. One team 3d printed 'adversarial examples of turtles'.
- **Algorithmic bias**

Limitations:

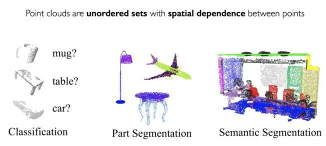
- data hungry
- Computationally intensive
- Adversarial examples
- Algorithmic bias
- Poor at representing uncertainty
- Uninterpretable black boxes, difficult to trust
- Difficult to encode structure and prior knowledge during learning
- Finicky to optimize:non-convex, choice of architecture
- Often require a lot of expert knowledge to design, fine tune architecture.

Convolutional neural networks were made to model the way images are thought to be process in the brain. We want to maintain spatial invariance in our convolutional neural networks.

6.1 Graphs as a structure for Representing Data

Convolutional networks have a convolutional kernel that picks up on what's inside. This process is based on elementwise multiplication and addition.By desiging these filters to particular types of weights we can pick up on different features that are in the image.

Graph Neural Networks have been used in molecular discovery. Message-passing neural networks have been deployed in molecular discovery to learn about bonds and relationships in molecular structure. A new antibiotic, Halicin was discovered with this method. Points clouds are unordered sets with spetial dependence between points


Point clouds are **unordered sets** with **spatial dependence** between points
mug?
table?
car?
Classification
Part Segmentation
Semantic Segmentation

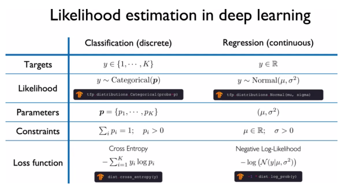
6.2 New Frontiers: Automated Machine Learning Autoencoders AI

It often requries experct knowledge to build an architecture for a given task. Can we build algorithms that learn which model to use to solve a given problem? This is the idea of AutoML.

7 Tuesday, January 26

Probabilistic learning is based on taking data, then a target, then making a prediction (this is the expected value). We also want to measure the uncertaintly (this is the variance) The task of classification is easy.

How do we learn continuous class targets? We'll assume that the distribution is continuous and in particular that it's Gaussian. Our prediction will be the expected value of the distribution.



Likelihood estimation in deep learning

	Classification (discrete)	Regression (continuous)
Targets	$y \in \{1, \cdots, K\}$	$y \in \mathbb{R}$
Likelihood	$y \sim \text{Categorical}(\boldsymbol{p})$ `tfp.distributions.Categorical(probs=p)`	$y \sim \text{Normal}(\mu, \sigma^2)$ `tfp.distributions.Normal(mu, sigma)`
Parameters	$\boldsymbol{p} = \{p_1, \cdots, p_K\}$	(μ, σ^2)
Constraints	$\sum_i p_i = 1; \quad p_i > 0$	$\mu \in \mathbb{R}; \quad \sigma > 0$
Loss function	Cross Entropy $-\sum_{i=1}^{K} y_i \log p_i$ `dist.cross_entropy(y)`	Negative Log-Likelihood $-\log\left(\mathcal{N}(y\|\mu, \sigma^2)\right)$ `-1 * dist.log_prob(y)`

Do not mistake likelihood for model confidence. There are several types of uncertainty.

- Aleatoric Uncertainty descibes the confidence in the input data. High when input data is noisy cannot be reduced by adding more data.
- Epistemic Uncertainty describes the confidence of the prediction. high when missing training data. can be reduced by adding more data.

Aleatoric uncertainty can be learning directly using neural networks. Epistemic uncertainty is much more challenging to estimate. How can a model undertand when it does not know the answer?

One solution: Don't train deterministic NN but instead train a Bayesian NN. Every time we feed in an input into the model, we get a slightly different model that alters based on weights, which themselves are created depending on a Bayesian distribution.

This is slow. It requires running the network T times for every input. Store T copies of the network in parallel. Sampling hinders real-time ability on edge devices Sensitive to choice of prior.

Suppose we have a model that wants to find the steering wheel rotation angle given the image of a car's view. Given the images, the σ^2 would represent the aleatoric (data) uncertainty. Over time, we aggregate a bunch of μs which we use to make our prediction. The epistemic uncertainty is $Var[\mu]$.

Sampling from an evidential distribution yields individual new distributions over the data

$$y \sim Normal(\mu, \sigma^2)$$

Assume the distribution parameters are not known, place priors over each andn probabilistically estimage.

$$\mu \sim Normal(\gamma, \sigma^2 v^{-1})$$

$$\sigma^2 \sim \Gamma^{-1}(\alpha, \beta)$$

$$\mu, \sigma^2 \sim NormalInvGamma(\gamma, \upsilon, \alpha, \beta)$$

We can also get the analog of evidential learning for classification. Continuous going back to discrete.

$$\int_\theta p(y|\theta)p(\theta|m)d\theta$$

$$||y - \mathbb{E}[\mu]||\Phi(m)$$

Maximize model fit $\iff$ error correction.

Evidential learning

7.1 Algorithmc Bias and Fairness

What does **bias** mean? You don't describe a watermelon as red watermelon but you describe a yellow watermelon as yellow watermelon.

Bias in image classification: a bride in the united states passed through a CNN. The output is "bride". A bride in India passed through a CNN, the output is "red", "costume", "performance art". There are several types of biases that can both be interpretation-driven or data-driven.