Neural Networks

Perceptron Algorithm

A perceptron is one of the simplest forms of artificial neural networks - imagine it as a basic decision-maker that draws a line to separate two groups of data points. Just like how you might draw a line to separate apples from oranges based on their features, a perceptron tries to find the best dividing line between two classes of data.

Technical Details

Basic Formula

The prediction ($\hat{y}$) is given by:

$\hat{y} = \text{step}(w\ x\ + w\ x\ + b)$

Where:

- (w , w) are weights
- (x , x) are input features
- b is the bias term
- step() is the step function

Update Rules

For a point with coordinates (p, q) and label y:

1. If correctly classified:
   - No changes needed to weights or bias
2. If classified positive but actually negative:
   - w = w - p
   - w = w - q
   - b = b -
3. If classified negative but actually positive:
   - w = w + p
   - w = w + q
   - b = b +

Where is the learning rate.

Key Points:

- Algorithm iteratively adjusts the decision boundary
- Learning continues until all points are correctly classified or maximum iterations reached
- Convergence guaranteed if data is linearly separable
- Visualized as a line in 2D space or hyperplane in higher dimensions

The final solution shows the optimal decision boundary (solid line), while the dotted lines show the algorithm's learning progression.

Log-Loss Error Function

Think of log-loss as a way to measure how wrong your predictions are in classification problems. Instead of just counting incorrect predictions, it punishes you more heavily when you're very confident about a wrong prediction (like being 99% sure someone is a fraud when they're not) and less when you're uncertain. It's like a teacher who takes off more points when you write a wrong answer confidently versus when you show some doubt.

Technical Details

Basic Formula

Log Loss = -1/N (y_i * log(p_i) + (1-y_i) * log(1-p_i))

Where:

- N is the number of observations
- y_i is the actual value (0 or 1)
- p_i is the predicted probability
- log is the natural logarithm

Properties

1. Always positive
2. Lower values are better
3. Perfect predictions → Log Loss = 0
4. Approaches infinity as predictions get worse

Key Points:

- Used primarily in binary classification
- Common in logistic regression
- Differentiable (good for gradient descent)
- Penalizes confident incorrect predictions heavily
- Rewards accurate probability estimates

Practical Applications:

- Machine learning model evaluation
- Model comparison
- Risk assessment
- Probability calibration
- Classification tasks

Log-loss is particularly useful when you need probabilistic predictions rather than just class labels.

Predictions

The prediction is essentially the answer we get from the algorithm:

- A discrete answer will be of the form “no” or “yes” (or 0 or 1).
- A continuous answer will be a number, normally between 0 and 1.

When our prediction gives us a continuous number between 0 and 1, this is equivalent to the probability that the given data point should belong to the given classification (e.g., a data point might get a prediction of 0.5, with this being equivalent to a 50% probability that it is correctly classified).

With our linear example, the probability is a function of the distance from the line. The further a data point is from the line, the larger the probability that it is classified correctly.

Sigmoid Functions

The way we move from discrete predictions to continuous is to simply change our activation function from a step function to a sigmoid function. A sigmoid function is an s-shaped function that gives us:

- Output values close to 1 when the input is a large positive number

- Output values close to 0 when the input is a large negative number
- Output values close to 0.5 when the input is close to zero

The formula for our sigmoid function is:

(x) = 1/(1 + e^(-x))

Before our model consisted of a line with a positive region and a negative region. But now, by applying the sigmoid function, our model consists of an entire probability space where, for each point, we can get a probability that the classification is correct.

Multi-Class Classification and Softmax

Imagine sorting different fruits into baskets - you don't just decide between apples and oranges, but also bananas, pears, and more. That's multi-class classification, and softmax is like your brain calculating the probability of which basket each fruit belongs in. Instead of a simple yes/no decision, it gives you percentage chances for each possible category.

Technical Details

Softmax Function

For a vector z of K classes:

softmax(z_i) = e^(z_i) / Σ(e^(z_j))

Where:

- z_i is the input for class i
- e is Euler's number
- Σ runs over all classes j

Properties

1. Outputs sum to 1 (100%)
2. All outputs are between 0 and 1
3. Larger inputs lead to larger probabilities
4. Preserves relative order of inputs

Key Features

- Converts raw scores to probabilities
- Handles any number of classes
- Each output represents probability for that class
- Commonly used in neural networks
- Differentiable (good for backpropagation)

Common Applications

- Image classification
- Natural language processing
- Speech recognition
- Document categorization
- Medical diagnosis

Loss Function

Usually paired with cross-entropy loss:

Loss = -Σ(y_i * log(p_i))

Where:

- y_i is true label (one-hot encoded)
- p_i is predicted probability

Softmax is essentially a "soft" version of the maximum function, giving proportional probabilities rather than a single winner.

One-Hot Encoding

Imagine you're organizing a wardrobe with different types of clothing items. Instead of labeling a piece as "number 1" or "number 2", you create separate yes/no columns for each type - one for shirts, one for pants, one for dresses, etc. That's one-hot encoding - converting categorical data into a format where each category gets its own binary column.

Technical Details

Basic Format

For categories [A, B, C]:

A → [1, 0, 0] B → [0, 1, 0] C → [0, 0, 1]

Example

Original data: "Color" column with [Red, Blue, Green]

Red → [1, 0, 0] Blue → [0, 1, 0] Green → [0, 0, 1]

Properties

- Only one '1' per encoding (hence "one-hot")
- Rest are '0's
- Number of columns equals number of categories
- No ordinal relationship implied
- Sparse representation

Common Applications

- Machine learning models
- Neural networks
- Text processing
- Categorical feature encoding
- Natural language processing

Advantages

- No ordinal relationship assumed
- Equal distance between categories
- Works well with most algorithms
- Clear binary representation

Disadvantages

- High dimensionality for many categories
- Sparse matrices
- Memory intensive
- Can be computationally expensive

One-hot encoding is essential when working with algorithms that expect numerical input but your data is categorical.

Maximum Likelihood

Imagine you're trying to figure out if a coin is fair by flipping it many times. Maximum likelihood is like asking: " What probability of getting heads would make my actual results most likely to occur?" It's a method that finds the parameters that make your observed data most probable.

The key idea is that we want to calculate P(all), which is the product of all the independent probabilities of each point. This helps indicate how well the model performs in classifying all the points. To get the best model, we will want to maximize this probability.

Technical Details

Basic Formula

Log Likelihood = Σ log(P(x|)) Where:

- P(x|) is probability of data x given parameters
- Σ sums over all observations
- Log is used for computational convenience

Key Concepts

1. Likelihood Function
   - Measures probability of observed data
   - Function of parameters, not data
   - Usually maximized using calculus
2. Log-Likelihood
   - Converts products to sums
   - Preserves same maximum
   - Computationally more stable

Common Applications

- Parameter estimation
- Statistical inference
- Machine learning models
- Distribution fitting
- Regression analysis

Steps

1. Write likelihood function
2. Take log of function
3. Find derivative
4. Set derivative to zero
5. Solve for parameters

Advantages

- Statistically well-founded
- Often has closed-form solution
- Provides consistent estimates
- Works with many distributions

Maximum likelihood gives us the parameters that make the observed data most probable, but doesn't guarantee they're the "true" parameters.

Cross-Entropy

The logarithm function has a very nice identity that says that the logarithm of a product is the sum of the logarithms of the factors:

log(ab) = log(a) + log(b)

So if we take the logarithm of our product, we get a sum of the logarithms of the factors.

We'll actually be taking the natural logarithm, ln, which is base e instead of base 10. In practice, everything works the same as what we showed here with log because everything gets scaled by the same factor. However, using ln is the convention, so we'll use it here as well.

The logarithm of a number between zero and one is always a negative number, so that means all of our probabilities (which are between zero and one) will give us negative results when we take their logarithm. Thus, we will want to take the negative of each of these results (i.e., multiply each one by -1).

In the end, what we calculate is a sum of negative logarithms of our probabilities, like this:

-log(0.6) - log(0.2) - log(0.1) - log(0.7) = 4.8

This is called the cross-entropy. A good model gives a high probability and the negative of a logarithm of a large number is a small number—thus, in the end:

- A high cross-entropy indicates a bad model
- A low cross-entropy indicates a good model

We can think of the negatives of these logarithms as errors at each point. The higher the probability, the lower the error—and the lower the cross-entropy. So now our goal has shifted from maximizing the probability to minimizing the cross-entropy.

Think of cross-entropy as a measure of how surprised your model is when seeing the true answers - like a test score where you get penalized more harshly for being confidently wrong than for being unsure. When you're very confident (90%) but wrong, you get a bigger penalty than when you're unsure (51%) and wrong.

Technical Details

Basic Formula

$H(y,\hat{y}) = -\Sigma\ y\_i * \log(\hat{y}\_i)$

Where:

- y_i are true values
- $\hat{y}\_i$ are predicted probabilities
- Σ sums over all classes

Formula for cross-entropy

Cross-entropy = $-\Sigma[y\_i \ln(p\_i) + (1 - y\_i)\ln(1 - p\_i)]$

From i=1 to m

Where:

- y_i is the true label (0 or 1)
- p_i is the predicted probability
- ln is the natural logarithm
- m is the number of training examples
- Σ represents the sum over all training examples

This is the binary cross-entropy formula specifically used for binary classification problems.

Key Properties

1. Always positive
2. Zero only when prediction equals reality
3. Higher when predictions are confidently wrong
4. Lower when predictions match true labels

Examples

- Perfect prediction (1.0) → CE 0
- Wrong prediction (0.0) → CE → ∞
- Uncertain prediction (0.5) → Moderate CE

Applications

- Classification problems
- Neural networks
- Model evaluation
- Information theory
- Deep learning

Important Points

- Used as loss function
- Measures prediction quality
- Related to maximum likelihood
- Good for gradient descent
- Works with multiple classes

Remember: The goal is to minimize cross-entropy, as lower values indicate better model predictions.

Multi-Class Cross-Entropy

Imagine a weather prediction system that needs to decide between sunny, rainy, or cloudy - not just a simple yes/no. Multi-class cross-entropy helps measure how well our predictions match reality when we have multiple possible outcomes, penalizing the model more heavily when it's confidently wrong about any of the classes.

Technical Details

Basic Formula

H(y,p) = -Σ Σ y_ij * log(p_ij)

Where:

- y_ij is 1 if sample i belongs to class j, else 0
- p_ij is predicted probability that sample i belongs to class j

- First Σ sums over all samples
- Second Σ sums over all classes

Properties

1. Extends binary cross-entropy to multiple classes
2. Works with one-hot encoded labels
3. Each class contributes to total loss
4. Always non-negative
5. Zero only for perfect predictions

Common Applications

- Deep learning classification
- Natural language processing
- Image recognition
- Speech recognition
- Document classification

Important Considerations

- Requires normalized probabilities (sum to 1)
- Often paired with softmax activation
- More computationally intensive than binary
- Handles class imbalance naturally
- Differentiable (good for gradient descent)

Remember: The goal remains minimizing the cross-entropy, which happens when predicted probabilities match true distributions across all classes.

Logistic Regression

Think of logistic regression like a sophisticated yes/no decision maker - similar to how a doctor might determine if a patient has a condition based on various symptoms. Despite its name, it's actually used for classification, not regression, and predicts the probability of an outcome being in a particular category.

Technical Details

Basic Formula

P(y=1) = 1 / (1 + e^(-z)) where z = wx + b

Where:

- w = weights
- x = input features
- b = bias term
- e = Euler's number

Key Components

1. Sigmoid Function
   - Transforms linear input to [0,1] range
   - Creates S-shaped curve
   - Output interpreted as probability
2. Cost Function (Binary Cross-Entropy)

$J(w,b) = -(1/m)\Sigma[y\_i \log(\hat{y}\_i) + (1-y\_i)\log(1-\hat{y}\_i)]$

Formula for the error function (for binary classification problems)

Error function $= -(1/m)\Sigma[(1 - y\_i)(\ln(1 - \hat{y}\_i)) + y\_i \ln(\hat{y}\_i)]$ From i=1 to m

And the total formula for the error is then:

$E(W,b) = -(1/m)\Sigma[(1 - y\_i)(\ln(1 - (Wx\hat{}(i) + b))) + y\_i \ln((Wx\hat{}(i) + b))]$ From i=1 to m

For multiclass problems, the error function is:

Error function $= -(1/m)\Sigma\ \Sigma\ y\_ij \ln(\hat{y}\_ij)$ From i=1 to m, j=1 to n

Now that we know how to calculate the error, our goal will be to minimize it.

Where:

- m is number of samples
- n is number of classes
- y_i are true values
- $\hat{y}$_i are predicted values
- W is weight matrix
- b is bias term
- is sigmoid function
- x^(i) is input vector for sample i

Properties

- Binary classification (usually)
- Outputs probabilities
- Requires numeric input features
- Assumes linear decision boundary
- Easy to interpret coefficients

Common Applications

- Medical diagnosis
- Credit risk assessment
- Email spam detection
- Customer churn prediction
- Marketing response prediction

Advantages

- Simple to implement
- Fast to train
- Probabilistic interpretation
- Works well with linear boundaries
- Less prone to overfitting

Gradient Descent

Imagine rolling a ball down a hill - it naturally finds the lowest point by following the steepest path downward. That's essentially what gradient descent does in machine learning: it finds the minimum of a function by repeatedly taking steps in the direction where the function decreases most quickly (the steepest descent).

The Calculation Process

Step 1: The Error Function

The error function E tells us how wrong our predictions are:

E = -(1/m)Σ(y_i ln(ŷ_i) + (1 - y_i)ln(1 - ŷ_i))

Step 2: Finding the Direction (Gradient)

We calculate partial derivatives to find which direction leads downhill:

- For weights: E/ w_j = -(y - ŷ)x_j
- For bias: E/ b = -(y - ŷ)

Step 3: Update Rule

We update parameters by moving in the opposite direction of the gradient:

w'_i ← w_i + (y - ŷ)x_i b' ← b + (y - ŷ)

Where:

- is the learning rate (step size)
- (y - ŷ) is the prediction error
- x_i are the input features

Significance

1. Larger errors cause bigger steps
2. Direction depends on whether prediction was too high or too low
3. Step size is controlled by learning rate
4. Process repeats until convergence

The beauty of this calculation is its elegant form: the gradient turns out to be just the error (y - ŷ) times the input features, making it computationally efficient and intuitively meaningful.

Gradient calculation

In the last few videos, we learned that in order to minimize the error function, we need to take some derivatives. So let's get our hands dirty and actually compute the derivative of the error function. The first thing to notice is that the sigmoid function has a really nice derivative. Namely,

'(x) = (x)(1 - (x))

The reason for this is the following, we can calculate it using the quotient formula:

'(x) = / x 1/(1+e^-x) = $e^{-x}/(1+e^{-x})^2$ = 1/(1+e^-x) · $e^{-x}/(1+e^{-x})$ = (x)(1 - (x))

And now, let's recall that if we have m points labelled x^(1), $x^{(2),\ldots,x}$(m), the error formula is:

E = -(1/m)Σ(y_i ln(ŷ_i) + (1 - y_i)ln(1 - ŷ_i))

where the prediction is given by ŷ_i = (Wx^(i) + b).

Our goal is to calculate the gradient of E, at a point x = (x ,…,x_n), given by the partial derivatives

E = (/ w E,…, / w_n E, / b E)

To simplify our calculations, we'll actually think of the error that each point produces, and calculate the derivative of this error. The total error, then, is the average of the errors at all the points. The error produced by each point is, simply,

$E = -y \ln(\hat{y}) - (1 - y)\ln(1 - \hat{y})$

In order to calculate the derivative of this error with respect to the weights, we'll first calculate $\partial / \partial w_j \hat{y}$. Recall that $\hat{y} = \sigma(Wx + b)$, so:

$\partial / \partial w_j \hat{y} = \partial / \partial w_j \sigma(Wx + b) = \sigma(Wx + b)(1 - \sigma(Wx + b)) \cdot \partial / \partial w_j(Wx + b) = \hat{y}(1 - \hat{y}) \cdot \partial / \partial w_j(Wx + b) = \hat{y}(1 - \hat{y}) \cdot \partial / \partial w_j(w_1 x_1 + \ldots + w_j x_j + \ldots + w_n x_n + b) = \hat{y}(1 - \hat{y}) \cdot x_j$

The last equality is because the only term in the sum which is not a constant with respect to w_j is precisely $w_j x_j$, which clearly has derivative x_j.

Now, we can go ahead and calculate the derivative of the error E at a point x, with respect to the weight w_j.

$\partial / \partial w_j E = \partial / \partial w_j[-y \log(\hat{y}) - (1 - y)\log(1 - \hat{y})] = -y \partial / \partial w_j \log(\hat{y}) - (1 - y) \partial / \partial w_j \log(1 - \hat{y}) = -y \cdot 1/\hat{y} \cdot \partial / \partial w_j \hat{y} - (1 - y) \cdot 1/(1-\hat{y}) \cdot \partial / \partial w_j(1 - \hat{y}) = -y \cdot 1/\hat{y} \cdot \hat{y}(1 - \hat{y})x_j - (1 - y) \cdot 1/(1-\hat{y}) \cdot (-1)\hat{y}(1 - \hat{y})x_j = -y(1 - \hat{y}) \cdot x_j + (1 - y)\hat{y} \cdot x_j = -(y - \hat{y})x_j$

A similar calculation will show us that

$\partial / \partial b E = -(y - \hat{y})$

This actually tells us something very important. For a point with coordinates $(x_1,\ldots,x_n)$, label y, and prediction $\hat{y}$, the gradient of the error function at that point is $(-(y - \hat{y})x_1,\ldots,-(y - \hat{y})x_n,-(y - \hat{y}))$. In summary, the gradient is

$\nabla E = -(y - \hat{y})(x_1,\ldots,x_n,1)$.

If you think about it, this is fascinating. The gradient is actually a scalar times the coordinates of the point! And what is the scalar? Nothing less than a multiple of the difference between the label and the prediction. What significance does this have?

So, a small gradient means we'll change our coordinates by a little bit, and a large gradient means we'll change our coordinates by a lot. If this sounds anything like the perceptron algorithm, this is no coincidence! We'll see it in a bit.

Gradient descent step

Therefore, since the gradient descent step simply consists in subtracting a multiple of the gradient of the error function at every point, then this updates the weights in the following way:

$w'_i \leftarrow w_i - \alpha[-(y - \hat{y})x_i]$,

which is equivalent to

$w'_i \leftarrow w_i + \alpha(y - \hat{y})x_i$.

Similarly, it updates the bias in the following way:

$b' \leftarrow b + \alpha(y - \hat{y})$,

Note: Since we've taken the average of the errors, the term we are adding should be $1/m \cdot \alpha$ instead of α, but as m is a constant, then in order to simplify calculations, we'll just take $1/m \cdot \alpha$ to be our learning rate, and abuse the notation by just calling it α.

Logistic Regression Algorithm

Think of logistic regression algorithm like teaching a computer to make yes/no decisions by gradually adjusting its decision-making weights until it gets better at predicting - similar to how you might adjust the temperature knob on an oven until you get it just right.

Algorithm Steps

Here are our steps for logistic regression:

1. Start with random weights: w ,…,w_n, b
2. For every point (x ,…,x_n):
    - For i = 1…n:
        - Update w'_i ← w_i - a(ŷ - y)x_i
        - Update b' ← b - a(ŷ - y)
3. Repeat until the error is small

Key Components

Initial Setup

- Begin with random weights and bias
- These are the starting parameters that will be refined

Update Process

- For each data point:
    - Calculate predicted value (ŷ)
    - Compare with actual value (y)
    - Adjust weights and bias accordingly

Convergence

- Continue updating until error becomes acceptably small
- Error measures the difference between predictions and actual values

Learning Rate (a)

- Controls how big steps we take in updating
- Too large: might overshoot
- Too small: slow learning

Here are the key points of contrast between gradient descent and the perceptron algorithm that Luis mentioned in the video:

Gradient Descent

With gradient descent, we change the weights from w_i to w_i + a(y - ŷ)x_i.

Perceptron Algorithm

With the perceptron algorithm we only change the weights on the misclassified points. If a point x is misclassified:

- We change w_i:
    - To w_i + ax_i if positive
    - To w_i - ax_i if negative
- If correctly classified: y - ŷ = 0
- If misclassified:

- y - $\hat{y}$ = 1 if positive
- y - $\hat{y}$ = -1 if negative

PERCEPTRON ALGORITHM:

If x is misclassified:

Change w_i to { w_i + x_i if positive w_i - x_i if negative }

If correctly classified: y-$\hat{y}$=0

If misclassified: { y-$\hat{y}$ = 1 if positive y-$\hat{y}$ = -1 if negative }

Neural Network Architecture

We will combine two linear models to get our non-linear model. Essentially the steps to do this are:

Calculate the probability for each model Apply weights to the probabilities Add the weighted probabilities Apply the sigmoid function to the result

Multiple layers Now, not all neural networks look like the one above. They can be way more complicated! In particular, we can do the following things:

Add more nodes to the input, hidden, and output layers. Add more layers. We'll see the effects of these changes in the next video.

Neural networks have a certain special architecture with layers:

The first layer is called the input layer, which contains the inputs. The next layer is called the hidden layer, which is the set of linear models created with the input layer. The final layer is called the output layer, which is where the linear models get combined to obtain a nonlinear model. Neural networks can have different architectures, with varying numbers of nodes and layers:

Input nodes. In general, if we have n nodes in the input layer, then we are modeling data in n-dimensional space (e.g., 3 nodes in the input layer means we are modeling data in 3-dimensional space). Output nodes. If there are more nodes in the output layer, this simply means we have more outputs—for example, we may have a multiclass classification model. Layers. If there are more layers then we have a deep neural network. Our linear models combine to create nonlinear models, which then combine to create even more nonlinear models!

Multi-Class Classification And here we elaborate a bit more into what can be done if our neural network needs to model data with more than one output.

When we have three or more classes, we could construct three separate neural networks—one for predicting each class. However, this is not necessary. Instead, we can add more nodes in the output layer. Each of these nodes will give us the probability that the item belongs to the given class.

Feedforward Neural Networks

Imagine a production line where raw materials (input) move through multiple processing stations, each station transforming the material a bit, until you get the final product (output). That's how feedforward works in neural networks - information flows forward through layers, each layer transforming the data in specific ways.

Technical Details

Process Steps

1. Take the input vector

2. Apply a sequence of linear models and sigmoid functions
3. Combine maps to create a highly non-linear map

Mathematical Formula

$\hat{y} =$ W^(2) W^(1)(x)

Where:

- is the sigmoid function
- W^(1) is first layer weights
- W^(2) is second layer weights
- represents function composition

Key Features

- One-way flow (forward only)
- Layer-by-layer processing
- Non-linear transformations
- Sequential computation
- No cycles or loops

Backpropagation

Think of backpropagation like tracing back your steps after making a mistake to figure out exactly where things went wrong. In neural networks, it's how we calculate which weights need adjusting and by how much, by working backwards from the output error to determine each layer's contribution to that error.

Now, we're ready to get our hands into training a neural network. For this, we'll use the method known as backpropagation. In a nutshell, backpropagation will consist of:

1. Doing a feedforward operation.
2. Comparing the output of the model with the desired output.
3. Calculating the error.
4. Running the feedforward operation backwards (backpropagation) to spread the error to each of the weights.
5. Use this to update the weights, and get a better model.
6. Continue this until we have a model that is good.

Sounds more complicated than what it actually is. Let's take a look in the next few videos. The first video will show us a conceptual interpretation of what backpropagation is.

Calculation of the derivative of the sigmoid function

Recall that the sigmoid function has a beautiful derivative, which we can see in the following calculation. This will make our backpropagation step much cleaner.

'(x) = / x $1/(1+e^{-x})$ = $e^{-x/(1+e}$-x)^2 = $1/(1+e^{-x})$ · $e^{-x/(1+e}$-x) = (x)(1 - (x))

Technical Details

Basic Process

1. Forward Pass:
   - Input goes through network
   - Calculate predicted output
   - Measure error

2. Backward Pass:
    - Start from output error
    - Calculate gradients layer by layer
    - Propagate error backwards

Key Components

Error gradient = Local gradient × Upstream gradient

Chain Rule Application

- Output layer: = (y - ŷ) × '(z)
- Hidden layers: = (W^T _next) × '(z) where:
- is error term
- ' is derivative of activation function
- W^T is transposed weight matrix

Update Rules

- Weights: W_new = W_old - × (input ×)
- Biases: b_new = b_old - ×

Key Features

- Efficient gradient computation
- Layer-wise updates
- Uses chain rule of calculus
- Enables deep learning
- Computationally efficient

Backpropagation is what makes deep learning possible by efficiently computing how each weight contributes to the overall error.

Implementing Gradient Descent

Mean Squared Error (MSE)

Imagine measuring how far off your darts are from the bullseye by measuring the distance of each throw, squaring those distances (to make all errors positive), and then taking the average. That's essentially what MSE does - it measures the average squared difference between predictions and actual values.

Technical Definition

MSE = (1/n) Σ(y_i - ŷ_i)²

Where:

- n is number of samples
- y_i is actual value
- ŷ_i is predicted value
- Σ sums over all samples

Key Properties

1. Always non-negative (due to squaring)
2. Perfect predictions yield MSE = 0

3. Larger errors are penalized more heavily
4. Unit of measurement is squared
5. Sensitive to outliers

Advantages

- Simple to compute
- Easy to differentiate
- Clear interpretation
- Good for regression problems
- Penalizes large errors more than small ones

Disadvantages

- Scale-dependent
- Can be dominated by outliers
- Squared units make interpretation less intuitive
- May not be ideal for classification tasks

MSE is particularly useful when larger errors are more problematic than smaller ones, as the squaring effect emphasizes bigger differences.

Gradient Descent with Squared Errors

CHECK EQ

We want to find the weights for our neural networks. Let's start by thinking about the goal. The network needs to make predictions as close as possible to the real values. To measure this, we use a metric of how wrong the predictions are, the error. A common metric is the sum of the squared errors (SSE):

$E = 1/2 \; \Sigma\Sigma[y\_j\hat{} \; - \hat{y}\_j\hat{} \;]^2$

where $\hat{y}$ is the prediction and y is the true value, and you take the sum over all output units j and another sum over all data points . This might seem like a really complicated equation at first, but it's fairly simple once you understand the symbols and can say what's going on in words.

First, the inside sum over j. This variable j represents the output units of the network. So this inside sum is saying for each output unit, find the difference between the true value y and the predicted value from the network $\hat{y}$, then square the difference, then sum up all those squares.

Then the other sum over is a sum over all the data points. So, for each data point you calculate the inner sum of the squared differences for each output unit. Then you sum up those squared differences for each data point. That gives you the overall error for all the output predictions for all the data points.

The SSE is a good choice for a few reasons. The square ensures the error is always positive and larger errors are penalized more than smaller errors. Also, it makes the math nice, always a plus.

Remember that the output of a neural network, the prediction, depends on the weights

$\hat{y}\_j\hat{} \; = f(\Sigma \; w\_ij \; x\_i\hat{} \;)$

and accordingly the error depends on the weights

$E = 1/2 \; \Sigma\Sigma[y\_j\hat{} \; - f(\Sigma \; w\_ij \; x\_i\hat{} \;)]^2$

We want the network's prediction error to be as small as possible and the weights are the knobs we can use to make that happen. Our goal is to find weights w_ij that minimize the squared error E. To do this with a neural network, typically you'd use gradient descent.

As Luis said, with gradient descent, we take multiple small steps towards our goal. In this case, we want to change the weights in steps that reduce the error. Continuing the analogy, the error is our mountain and we want to get to the bottom. Since the fastest way down a mountain is in the steepest direction, the steps taken should be in the direction that minimizes the error the most. We can find this direction by calculating the gradient of the squared error.

Gradient is another term for rate of change or slope. If you need to brush up on this concept, check out Khan Academy's on the topic.

To calculate a rate of change, we turn to calculus, specifically derivatives. A derivative of a function f(x) gives you another function f'(x) that returns the slope of f(x) at point x. For example, consider $f(x) = x^2$. The derivative of x^2 is $f'(x) = 2x$. So, at $x = 2$, the slope is $f'(2) = 4$. Plotting this out, it looks like:

The gradient is just a derivative generalized to functions with more than one variable. We can use calculus to find the gradient at any point in our error function, which depends on the input weights. You'll see how the gradient descent step is derived on the next page.

Below I've plotted an example of the error of a neural network with two inputs, and accordingly, two weights. You can read this like a topographical map where points on a contour line have the same error and darker contour lines correspond to larger errors.

At each step, you calculate the error and the gradient, then use those to determine how much to change each weight. Repeating this process will eventually find weights that are close to the minimum of the error function, the black dot in the middle.

Since the weights will just go wherever the gradient takes them, they can end up where the error is low, but not the lowest. These spots are called local minima. If the weights are initialized with the wrong values, gradient descent could lead the weights into a local minimum, illustrated below.

If we want our neural network to make reasonable predictions, we need to have a way of setting the weights. To address this, we can present the neural network with data that we know to be true and then set the model parameters (the weights) to match that data.

The most essential component we need for this is some measure of how bad our predictions are. The measure we'll use is the sum of the squared errors (SSE), which looks like this:

$$E = 1/2 \, \Sigma(\hat{y} - \hat{\hat{y}})^2$$

The SSE is a measure of our network's performance. If it's high, the network is making bad predictions. If it's low, the network is making good predictions. Minimizing the SSE is our goal in gradient descent:

- Starting at some random weight, we make a step in the direction towards the minimum, opposite to the gradient or slope.
- If we take many steps, always descending down a gradient, eventually the weights will find the minimum of the error function.

Mean Square Error

We're going to make a small change to how we calculate the error here. Instead of the SSE, we're going to use the mean of the square errors (MSE). Now that we're using a lot of data, summing up all the weight steps can lead to really large updates that make the gradient descent diverge. To compensate for this, you'd need to use a quite small learning rate. Instead, we can just divide by the number of records in our data, m to take the average. This way, no matter how much data we use, our learning rates will typically be in the range of 0.01 to 0.001. Then, we can use the MSE (shown below) to calculate the gradient and the result is the same as before, just averaged instead of summed.

$$E = 1/2m \, \Sigma(\hat{y} - \hat{\hat{y}})^2$$

Here's the general algorithm for updating the weights with gradient descent:

- Set the weight step to zero: $\Delta w_i = 0$
- For each record in the training data:
  - Make a forward pass through the network, calculating the output $\hat{y} = f(\Sigma_i w_i x_i)$
  - Calculate the error term for the output unit, $= (y - \hat{y}) * f'(\Sigma_i w_i x_i)$
  - Update the weight step $\Delta w_i = \Delta w_i + x_i$
- Update the weights $w_i = w_i + \Delta w_i/m$ where is the learning rate and m is the number of records. Here we're averaging the weight steps to help reduce any large variations in the training data.
- Repeat for e epochs.

You can also update the weights on each record instead of averaging the weight steps after going through all the records.

Remember that we're using the sigmoid for the activation function, f(h) = 1/(1 + e^-h)

And the gradient of the sigmoid is f'(h) = f(h)(1 - f(h))

where h is the input to the output unit,

h = Σ_i w_ix_i

Backpropagation

Now we've come to the problem of how to make a multilayer neural network learn. Before, we saw how to update weights with gradient descent. The backpropagation algorithm is just an extension of that, using the chain rule to find the error with the respect to the weights connecting the input layer to the hidden layer (for a two layer network).

To update the weights to hidden layers using gradient descent, you need to know how much error each of the hidden units contributed to the final output. Since the output of a layer is determined by the weights between layers, the error resulting from units is scaled by the weights going forward through the network. Since we know the error at the output, we can use the weights to work backwards to hidden layers.

For example, in the output layer, you have errors attributed to each output unit k. Then, the error attributed to hidden unit j is the output errors, scaled by the weights between the output and hidden layers (and the gradient):

$= W \quad f'(h)$

Then, the gradient descent step is the same as before, just with the new errors:

$\Delta w = x$

where w are the weights between the inputs and hidden layer and x are input unit values. This form holds for however many layers there are. The weight steps are equal to the step size times the output error of the layer times the values of the inputs to that layer

$\Delta w\ q =$ output V

Here, you get the output error, output, by propagating the errors backwards from higher layers. And the input values, V are the inputs to the layer, the hidden layer activations to the output unit for example.

Let's walk through the steps of calculating the weight updates for a simple two layer network. Suppose there are two input values, one hidden unit, and one output unit, with sigmoid activations on the hidden and output units. The following image depicts this network. (Note: the input values are shown as nodes at the bottom of the image, while the network's output value is shown as $\hat{y}$ at the top. The inputs themselves do not count as a layer, which is why this is considered a two layer network.)

[Network diagram showing weights and connections]

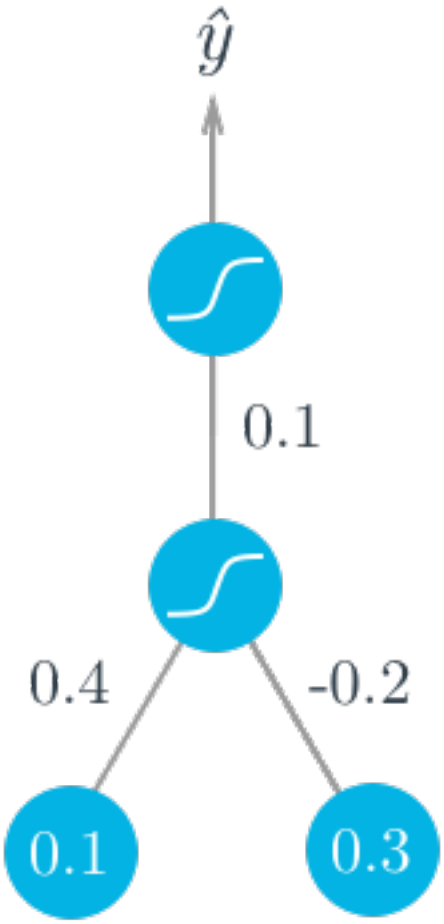

Figure 1: title

Assume we're trying to fit some binary data and the target is y = 1. We'll start with the forward pass, first calculating the input to the hidden unit

h = w x = 0.1 × 0.4 - 0.2 × 0.3 = -0.02

and the output of the hidden unit

a = f(h) = sigmoid(-0.02) = 0.495.

Using this as the input to the output unit, the output of the network is

$\hat{y}$ = f(W · a) = sigmoid(0.1 × 0.495) = 0.512.

With the network output, we can start the backwards pass to calculate the weight updates for both layers. Using the fact that for the sigmoid function f'(W · a) = f(W · a)(1 - f(W · a)), the error term for the output unit is

= (y - $\hat{y}$)f'(W · a) = (1 - 0.512) × 0.512 × (1 - 0.512) = 0.122.

Now we need to calculate the error term for the hidden unit with backpropagation. Here we'll scale the error term from the output unit by the weight W connecting it to the hidden unit. For the hidden unit error term, = W f'(h), but since we have one hidden unit and one output unit, this is much simpler.

= W f'(h) = 0.1 × 0.122 × 0.495 × (1 - 0.495) = 0.003

Now that we have the errors, we can calculate the gradient descent steps. The hidden to output weight step is the learning rate, times the output unit error, times the hidden unit activation value.

ΔW = a = 0.5 × 0.122 × 0.495 = 0.0302

Then, for the input to hidden weights w , it's the learning rate times the hidden unit error, times the input values.

Δw = x = (0.5 × 0.003 × 0.1, 0.5 × 0.003 × 0.3) = (0.00015, 0.00045)

From this example, you can see one of the effects of using the sigmoid function for the activations. The maximum derivative of the sigmoid is 0.25, so the errors in the output layer get reduced by at least 75%, and errors in the hidden layer are scaled down by at least 93.75%! You can see that if you have a lot of layers, using a sigmoid activation function will quickly reduce the weight steps to tiny values in layers near the input. This is known as the vanishing gradient problem. Later in the course you'll learn about other activation functions that perform better in this regard and are more commonly used in modern network architectures.

Now we've seen that the error term for the output layer is

= (y - $\hat{y}$)f'(a)

and the error term for the hidden layer is

= [w]f'(h)

For now we'll only consider a simple network with one hidden layer and one output unit. Here's the general algorithm for updating the weights with backpropagation:

• Set the weight steps for each layer to zero • The input to hidden weights Δw = 0 • The hidden to output weights ΔW = 0

• For each record in the training data: • Make a forward pass through the network, calculating the output $\hat{y}$ • Calculate the error gradient in the output unit, = (y - $\hat{y}$)f'(z) where z = W a , the input to the output unit. • Propagate the errors to the hidden layer = W f'(h) • Update the weight steps: • ΔW = ΔW + a • Δw = Δw + a

• Update the weights, where is the learning rate and m is the number of records: • W = W + ΔW /m • w = w + Δw /m • Repeat for e epochs.

Training Neural Networks

We've learned how to build a deep neural network and how to train it to fit our data. However, there are many things that can fail when training a neural network. For example:

Our architecture can be poorly chosen Our data can be noisy Our model can take far too long to run

We need to learn ways to optimize the training of our models—and that's what we'll be getting into in this lesson! By the end of this lesson, you'll be able to:

Separate data into testing and training sets in order to objectively test a model and ensure that it can generalize beyond the training data. Distinguish between underfitting and overfitting, and identify the underlying causes of each. Use early stopping to end the training process at a point that minimizes both testing error and training error. Apply regularization to reduce overfitting. Use dropout to randomly turn off portions of a network during training and ensure no single part of the network dominates the resulting model disproportionately. Use random restart to avoid getting stuck in local minima. Use the hyperbolic tangent function and ReLU to improve gradient descent. Distinguish between batch gradient descent vs stochastic gradient descent. Adjust the learning rate of the gradient descent algorithm in order to improve model optimization. Use momentum to avoid getting stuck in local minima.

When comparing models to determine which is better, we need a way of objectively testing them. To accomplish this, we can divide our data into two parts:

A training set that we use to train the models A testing set that we use only for testing the models While training, we only use the training data—we set the testing data aside and don't use it as input for our learning algorithm.

Then, once our models are trained, we reintroduce the testing set. A model that performs well on the training data may not perform well on the testing data. We'll see one reason for this, called overfitting, next.

When we train our models, it is entirely possible to get them to a point where they perform very well on our training data—but then perform very poorly on our testing data. Two common reasons for this are underfitting and overfitting

Underfitting

Underfitting means that our model is too simplistic. There is a poor fit between our model and the data because we have oversimplified the problem. Underfitting is sometimes referred to as error due to bias. Our training data may be biased and this bias may be incorporated into the model in a way that oversimplifies it. For example, suppose we train an image classifier to recognize dogs. And suppose that the only type of animal in the training set is a dog. Perhaps the model learns a biased and overly simple rule like, "if it has four legs it is a dog". When we then test our model on some data that has other animals, it may misclassify a cat as a dog—in other words, it will underfit the data because it has error due to bias.

Overfitting

Overfitting means that our model is too complicated. The fit between our model and the training data is too specific—the model will perform very well on the training data but will fail to generalize to new data. Overfitting is sometimes referred to as error due to variance. This means that there are random or irrelevant differences among the data points in our training data and we have fit the model so closely to these irrelevant differences that it performs poorly when we try to use it with our testing data. For example, suppose we want our image classifier to recognize dogs, but instead we train it to recognize "dogs that are yellow, orange, or grey." If our testing set includes a dog that is brown, for example, our model will put it in a separate class, which was not what we wanted. Our model is too specific—we have fit the data to some unimportant differences in the training data and now it will fail to generalize.

Applying This to Neural Networks Generally speaking, underfitting tends to happen with neural networks that have overly simple architecture, while overfitting tends to happen with models that are highly complex.

The bad news is, it's really hard to find the right architecture for a neural network. There is a tendency to create a network that either has overly simplistic architecture or overly complicated architecture. In general terms, the approach we will take is to err on the side of an overly complicated model, and then we'll apply certain techniques to reduce the risk of overfitting.

Early Stopping

When training our neural network, we start with random weights in the first epoch and then change these weights as we go through additional epochs. Initially, we expect these changes to improve our model as the neural network fits the training data more closely. But after some time, further changes will start to result in overfitting.

We can monitor this by measuring both the training error and the testing error. As we train the network, the training error will go down—but at some point, the testing error will start to increase. This indicates overfitting and is a signal that we should stop training the network prior to that point. We can see this relationship in a model complexity graph like this one:

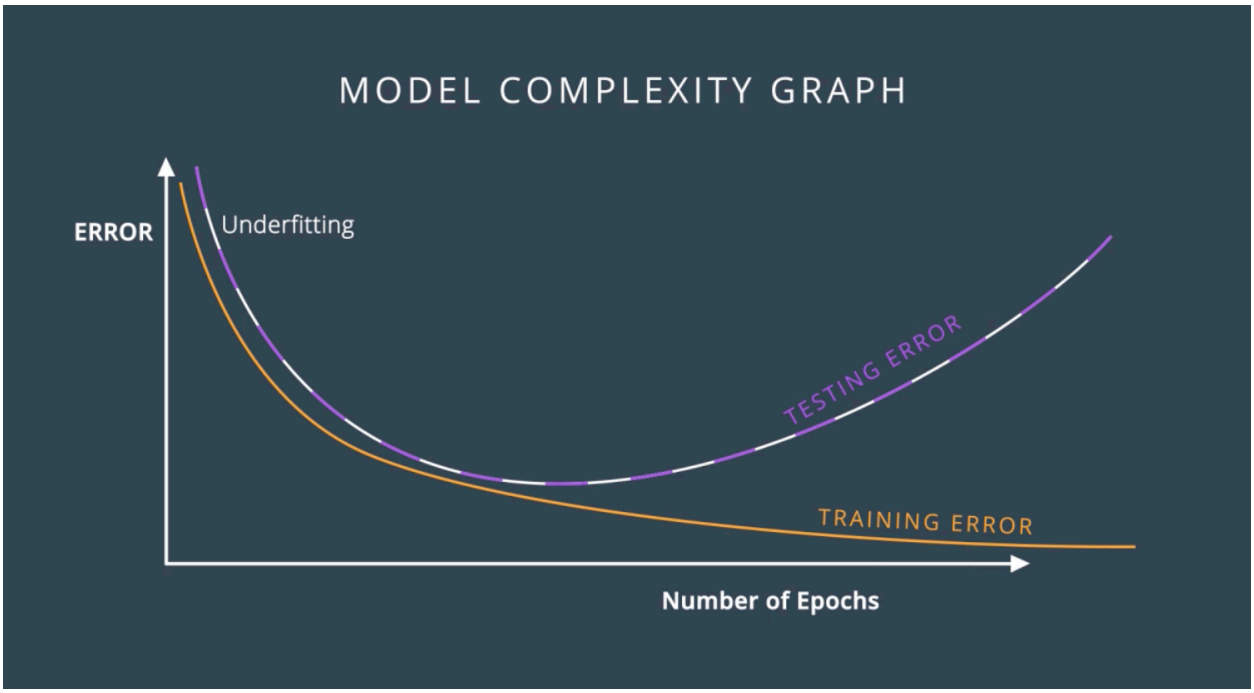

Figure 2: text

Have a look at the graph and make sure you can recognize the following:

On the Y-axis, we have a measure of the error and on the X-axis we have a measure of the complexity of the model (in this case, it's the number of epochs). On the left we have high testing and training error, so we're underfitting. On the right, we have high testing error and low training error, so we're overfitting. Somewhere in the middle, we have our happy Goldilocks point (the point that is "just right"). In summary, we do gradient descent until the testing error stops decreasing and starts to increase. At that moment, we stop. This algorithm is called early stopping and is widely used to train neural networks.

Large Co-efficients -> Overfitting Small Co-efficients -> Underfitting

Considering the Activation Functions

A key point here is to consider the activation functions of these two equations:

Solution: Regularization

LARGE COEFFICIENTS ⟶ OVERFITTING

PENALIZE LARGE WEIGHTS

$$(w_1, \ldots, w_n)$$

$$\text{ERROR FUNCTION} = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i)+y_i ln(\hat{y}_i)+\boxed{\lambda(|w_1|+\ldots+|w_n|)}$$

$$\text{ERROR FUNCTION} = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i)+y_i ln(\hat{y}_i)+\boxed{\lambda({w_1}^2+\ldots+{w_n}^2)}$$

Figure 3: text

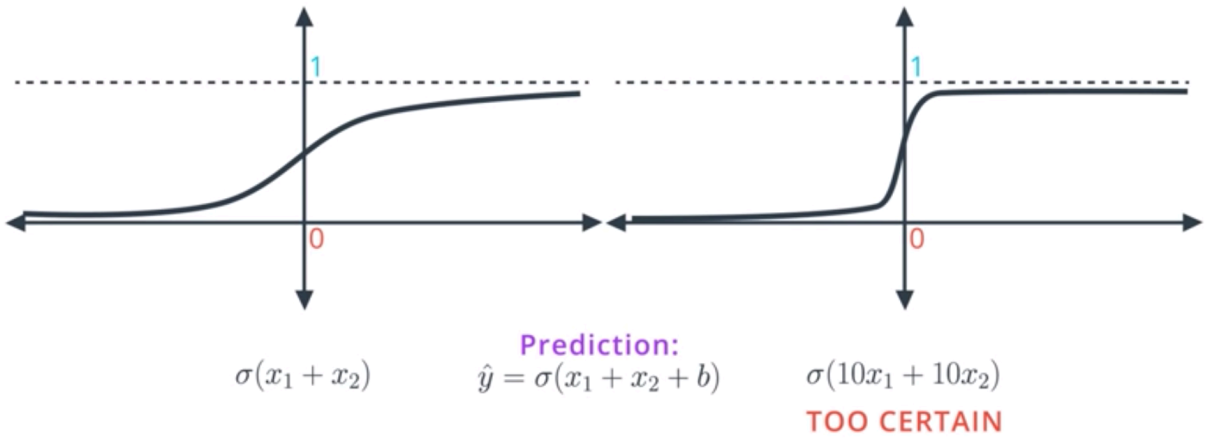

Figure 4: text

When we apply sigmoid to small values such as x1 + x2, we get the function on the left, which has a nice slope for gradient descent. When we multiply the linear function by 10 and take sigmoid of 10x1 + 10x2 , our predictions are much better since they're closer to zero and one. But the function becomes much steeper and it's much harder to do graident descent.

Conceptually, the model on the right is too certain and it gives little room for applying gradient descent. Also, the points that are classified incorrectly in the model on the right will generate large errors and it will be hard to tune the model to correct them.

Regularization

Now the question is, how do we prevent this type of overfitting from happening? The trouble is that large coefficients are leading to overfitting, so what we need to do is adjust our error function by, essentially, penalizing large weights.

If you recall, our original error function looks like this:

$-1/m \qquad (1 - y)\ln(1 - \hat{y}) + y \ln(\hat{y})$

We want to take this and add a term that is big when the weights are big. There are two ways to do this. One way is to add the sums of absolute values of the weights times a constant lambda:

$+ (|w| + \ldots + |w|)$

The other one is to add the sum of the squares of the weights times that same constant:

$+ (w^2 + \ldots + w^2)$

In both cases, these terms are large if the weights are large.

L1 vs L2 Regularization The first approach (using absolute values) is called L1 regularization, while the second (using squares) is called L2 regularization. Here are some general guidelines for deciding between the two:

L1 Regularization L1 tends to result in sparse vectors. That means small weights will tend to go to zero. If we want to reduce the number of weights and end up with a small set, we can use L1. L1 is also good for feature selection. Sometimes we have a problem with hundreds of features, and L1 regularization will help us select which ones are important, turning the rest into zeroes.

L2 Regularization L2 tends not to favor sparse vectors since it tries to maintain all the weights homogeneously small. L2 gives better results for training models so it's the one we'll use the most.

Activation Functions

Think of activation functions as the decision-makers in neural networks - they determine whether and how strongly each neuron should "fire" based on its inputs. Just like neurons in our brain don't fire in a simple linear way, these functions introduce non-linearity that allows neural networks to learn complex patterns.

Mathematically, activation functions transform the output of each neuron:

Sigmoid: $(x) = 1/(1 + e\ )$

- Output range: (0,1)
- Gradient: $f'(x) = (x)(1 - (x))$
- Key issues:
  - Saturating gradients: $f'(x) \to 0$ as $|x| \to \infty$
  - Not zero-centered
  - Computationally expensive

ReLU: $f(x) = \max(0,x)$

- Output range: [0,∞)
- Gradient: f'(x) = {1 if x > 0, 0 otherwise}
- Benefits:
  - No saturation for positive values
  - Sparse activation (~50%)
  - Computationally efficient
- Issue: "Dying ReLU" problem

Modern Variants:

1. Leaky ReLU: f(x) = {x if x > 0, x otherwise}
2. ELU: f(x) = {x if x > 0, (e^x - 1) otherwise}
3. SELU: Self-normalizing variant

Regularization

Think of regularization as putting a leash on your neural network - it prevents the model from becoming too complex and "memorizing" the training data. Just like how we want students to learn general concepts rather than memorize specific examples, regularization helps neural networks learn patterns that generalize well to new data.

Mathematically, regularization modifies the loss function L( ) by adding penalty terms:

L Regularization: L_reg( ) = L( ) + |w |

- Gradient: Ω/ w = sign(w)
- Properties:
  - Creates sparse solutions (many w = 0)
  - Effective feature selection
  - Non-differentiable at w = 0

L Regularization: L_reg( ) = L( ) + w ²

- Gradient: Ω/ w = 2w
- Properties:
  - Weight decay interpretation: w ← w(1 - 2) - L/ w
  - Smoother solutions
  - All weights shrink proportionally

Combined Approaches:

- Elastic Net: |w | + (1-) w ²
- Dropout: Randomly zero out units (p: keep probability) Training: y = f(Wx) * mask/p Inference: y = f(Wx)
- Early Stopping: Monitor validation error

The strength parameter controls the trade-off between fitting the data and keeping weights small.

Dropout

When training a neural network, sometimes one part of the network has very large weights and it ends up dominating the training, while another part of the network doesn't really play much of a role (so it doesn't get trained).

To solve this, we can use a method called dropout in which we turn part of the network off and let the rest of the network train:

We go through the epochs and randomly turn off some of the nodes. This forces the other nodes to pick up the slack and take a larger part in the training. To drop nodes, we give the algorithm a parameter that indicates the probability that each node will get dropped during each epoch. For example, if we set this

parameter to 0.2, this means that during each epoch, each node has a 20% probability of being turned off. Note that some nodes may get turned off more than others and some may never get turned off. This is OK since we're doing it over and over; on average, each node will get approximately the same treatment.

Local Minima and Random Restart

The optimization landscape of neural networks is incredibly complex, with many peaks, valleys, and saddle points. When training gets stuck in a poor local minimum, we might end up with a suboptimal solution despite the network appearing to be "trained". Random restart is one of the fundamental strategies to escape these local minima.

Mathematically, the challenge can be formalized as:

Local Minimum Problem:

- For weight vector w, a local minimum occurs when:
  - L(w) = 0 (gradient is zero)
  - ^{2}L(w) is positive definite (all eigenvalues > 0)
- Not all local minima are created equal: L(w_local) > L(w_global)

Random Restart Techniques:

1. Basic Random Restart:
   - Initialize: w_new = w_0 + , ~ N(0, ^{2})
   - Run multiple times with different seeds
   - Keep best solution: w* = argmin_w L(w)
2. Momentum-Based Escape:
   - Update rule: v_t = v_(t-1) - L(w)
   - Weight update: w_t = w_(t-1) + v_t
   - : momentum coefficient
3. Learning Rate Scheduling:
   - _t = _0 / √t
   - Allows larger steps early, finer tuning later

Practical Considerations:

- Number of restarts needed increases with dimension
- Trade-off between exploration (high ^{2}) and exploitation
- Computational cost vs. solution quality
- Can combine with other techniques (e.g., simulated annealing)

Other Activation Functions

The Vanishing Gradient Problem To summarize the vanishing gradient problem:

- The sigmoid curve gets pretty flat on the sides, so if we calculate the derivative at a point far to the right or far to the left, this derivative is almost zero.
- This is problematic because it is the derivative that tells us what direction to move in. This is especially problematic in most linear perceptrons.

Hyperbolic Tangent

One alternative activation function is the hyperbolic tangent, which is given by the formula:

tanh(x) = (e^x - e$^{(-x))/(e}$x + e^(-x))

This is similar to sigmoid, but since our range is between minus one and one, the derivatives are larger—and this small difference has a big impact on neural networks.

Rectified Linear Unit (ReLU)

Another very popular activation function is the Rectified Linear Unit (ReLU). The formula is:

relu(x) = {x if x 0 0 if x < 0

In other words: • If the input is positive, return the same value. • If the input is negative, return zero.

This function is used a lot instead of the sigmoid and it can improve the training significantly without sacrificing much accuracy (since the derivative is one if the number is positive).

Batch Gradient Descent

First, let's review our batch gradient descent algorithm:

In order to decrease error, we take a bunch of steps following the negative of the gradient, which is the error function. Each step is called an epoch. In each epoch, we take our input (all of our data) and run it through the entire neural network. Then we find our predictions and calculate the error (how far the predictions are from the actual labels). Finally, we back-propagate this error in order to update the weights in the neural network. This will give us a better boundary for predicting our data. If we have a large number of data points then this process will involve huge matrix computations, which would use a lot of memory.

Stochastic Gradient Descent

To expedite this, we can use only some of our data at each step. If the data is well-distributed then a subset of the data can give us enough information about the gradient.

This is the idea behind stochastic gradient descent. We take small subsets of the data and run them through the neural network, calculating the gradient of the error function based on these points and moving one step in that direction.

We still want to use all our data, so what we do is the following:

Split the data into several batches. Take the points in the first batch and run them through the neural network. Calculate the error and its gradient. Back-propagate to update the weights (which will define a better boundary region). Repeat the above steps for the rest of the batches. Notice that with this approach we take multiple steps, whereas with batch gradient descent we take only one step with all the data. Each step may be less accurate, but, in practice, it's much better to take a bunch of slightly inaccurate steps than to take only one good one.

Learning Rate Decay Here are some key ideas to keep in mind when choosing a learning rate:

If the learning rate is large:

This means the algorithm is taking large steps, which can make it faster. However, the large steps may cause it to miss (overshoot) the minimum. If the learning learning rate is small:

This means the algorithm is taking small steps, which can make it slower. However, it will make steady progress and have a better chance of arriving at the local minimum. If your model is not working, a good general rule of thumb is to try decreasing the learning rate. The best learning rates are those that decrease as the algorithm is getting closer to a solution.

Momentum

Another way to solve the local minimum problem is with momentum. Momentum is a constant between 0 and 1.

We use to get a sort of weighted average of the previous steps:

step(n) + step(n - 1) + ^{2}step(n - 2) + ^{3}step(n - 3) + …

The previous step gets multiplied by 1, the one before it gets multiplied by , the one before that by ², the one before that by ³, and so on. Because has a value between 0 and 1, raising it to increasingly large powers means that the value will get smaller and smaller. In this way, the steps that happened a long time ago will be multiplied by tiny values and thus matter less than the ones that happened recently.

This can get us over "humps" and help us discover better minima. Once we get to the global minimum, the momentum will still be pushing us away, but not as much.

In this lesson, we learned ways to optimize the training of our models. If you followed along with everything, you now know how to:

- Separate data into testing and training sets in order to objectively test a model and ensure that it can generalize beyond the training data.
- Distinguish between underfitting and overfitting, and identify the underlying causes of each.
- Use early stopping to end the training process at a point that minimizes both testing error and training error.
- Apply regularization to reduce overfitting.
- Use dropout to randomly turn off portions of a network during training and ensure no single part of the network dominates the resulting model disproportionately.
- Use random restart to avoid getting stuck in local minima.
- Use the hyperbolic tangent function and ReLU to improve gradient descent.
- Distinguish between batch gradient descent vs stochastic gradient descent. Adjust the learning rate of the gradient descent algorithm in order to improve model optimization. Use momentum to avoid getting stuck in local minima.

Transfer Learning

The Four Main Cases When Using Transfer Learning Transfer learning involves taking a pre-trained neural network and adapting the neural network to a new, different data set.

Depending on both:

- the size of the new data set, and
- the similarity of the new data set to the original data set

the approach for using transfer learning will be different. There are four main cases:

1. new data set is small, new data is similar to original training data
2. new data set is small, new data is different from original training data
3. new data set is large, new data is similar to original training data
4. new data set is large, new data is different from original training data

A large data set might have one million images. A small data could have two-thousand images. The dividing line between a large data set and small data set is somewhat subjective. Overfitting is a concern when using transfer learning with a small data set.

Images of dogs and images of wolves would be considered similar; the images would share common characteristics. A data set of flower images would be different from a data set of dog images.

Each of the four transfer learning cases has its own approach. In the following sections, we will look at each case one by one.

Demonstration Network

To explain how each situation works, we will start with a generic pre-trained convolutional neural network and explain how to adjust the network for each case. Our example network contains three convolutional layers and three fully connected layers:

Here is an generalized overview of what the convolutional neural network does:

- the first layer will detect edges in the image

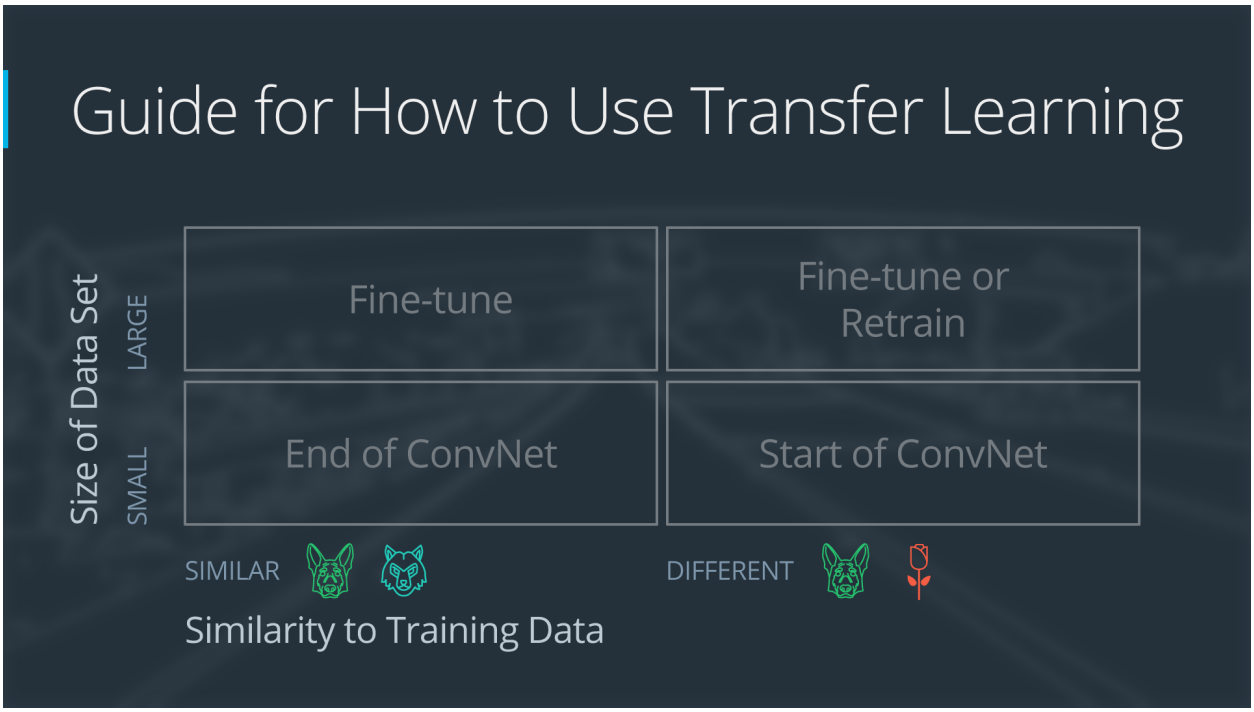

Figure 5: title

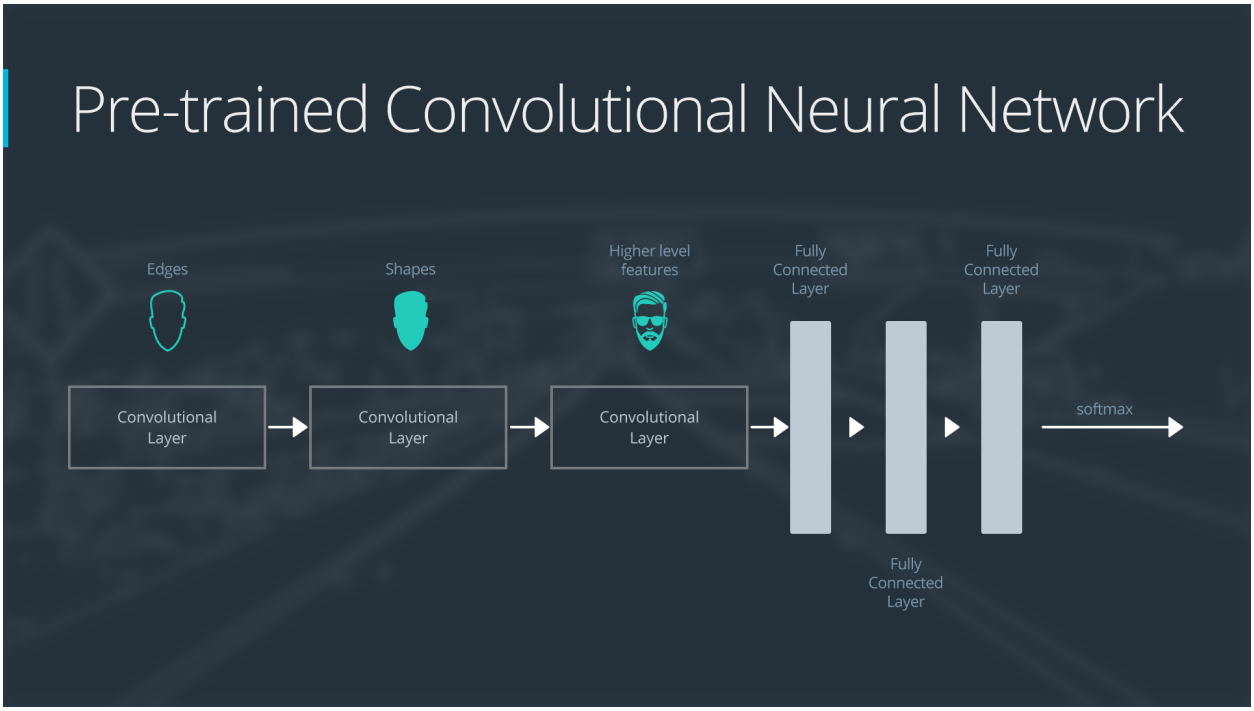

Figure 6: title

- the second layer will detect shapes
- the third convolutional layer detects higher level features

Each transfer learning case will use the pre-trained convolutional neural network in a different way.

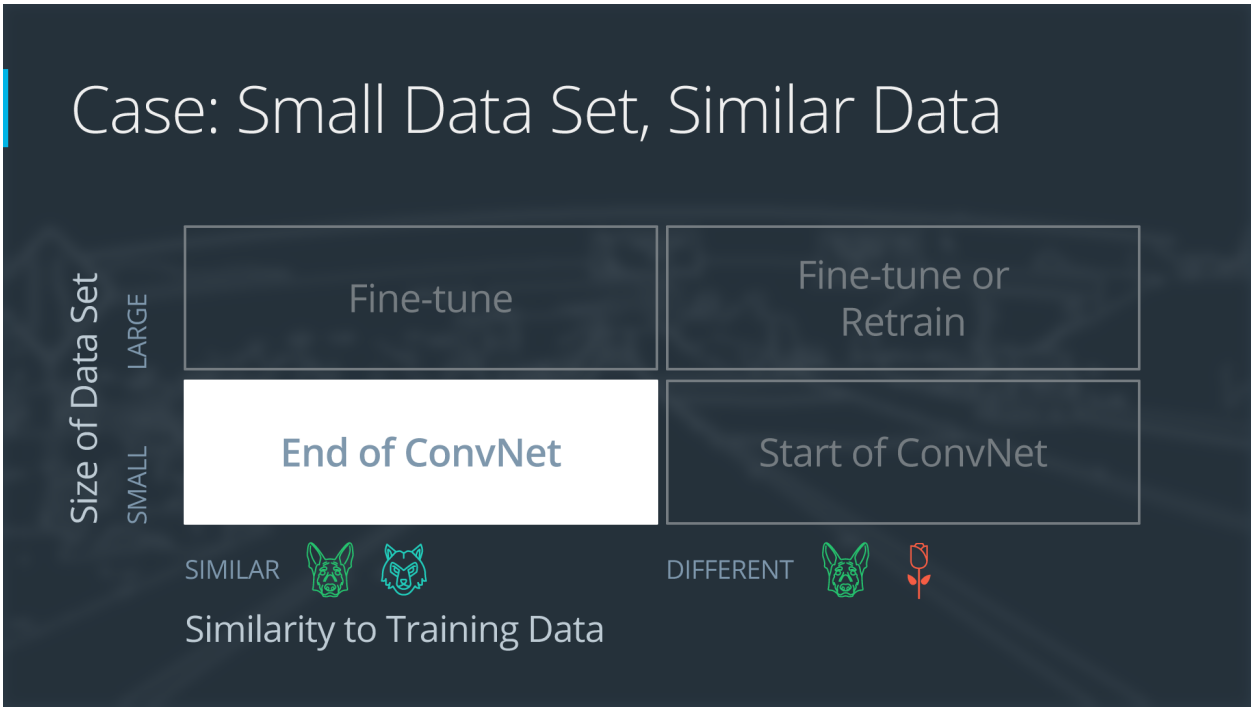

Figure 7: title

Case 1: Small Data Set, Similar Data If the new data set is small and similar to the original training data:

- slice off the end of the neural network
- add a new fully connected layer that matches the number of classes in the new data set
- randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network
- train the network to update the weights of the new fully connected layer

To avoid overfitting on the small data set, the weights of the original network will be held constant rather than re-training the weights.

Since the data sets are similar, images from each data set will have similar higher level features. Therefore most or all of the pre-trained neural network layers already contain relevant information about the new data set and should be kept.

Here's how to visualize this approach:

Case 2: Small Data Set, Different Data If the new data set is small and different from the original training data:

- slice off most of the pre-trained layers near the beginning of the network
- add to the remaining pre-trained layers a new fully connected layer that matches the number of classes in the new data set
- randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network

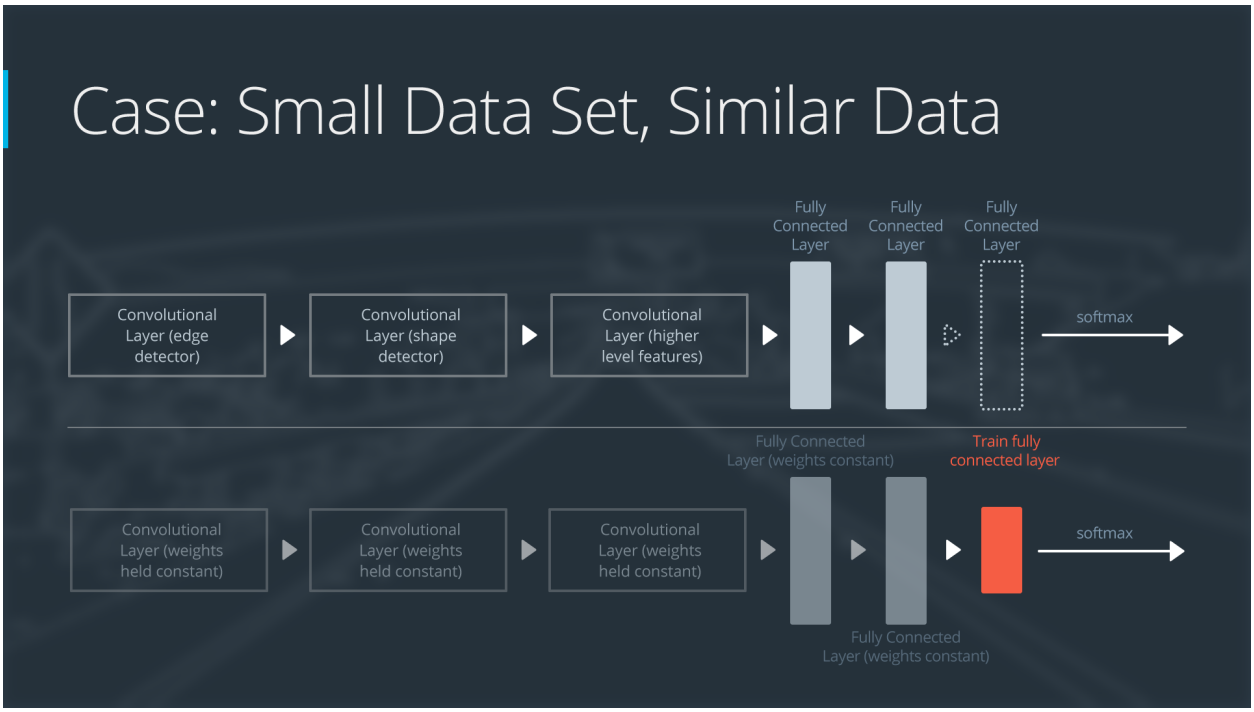

Figure 8: title

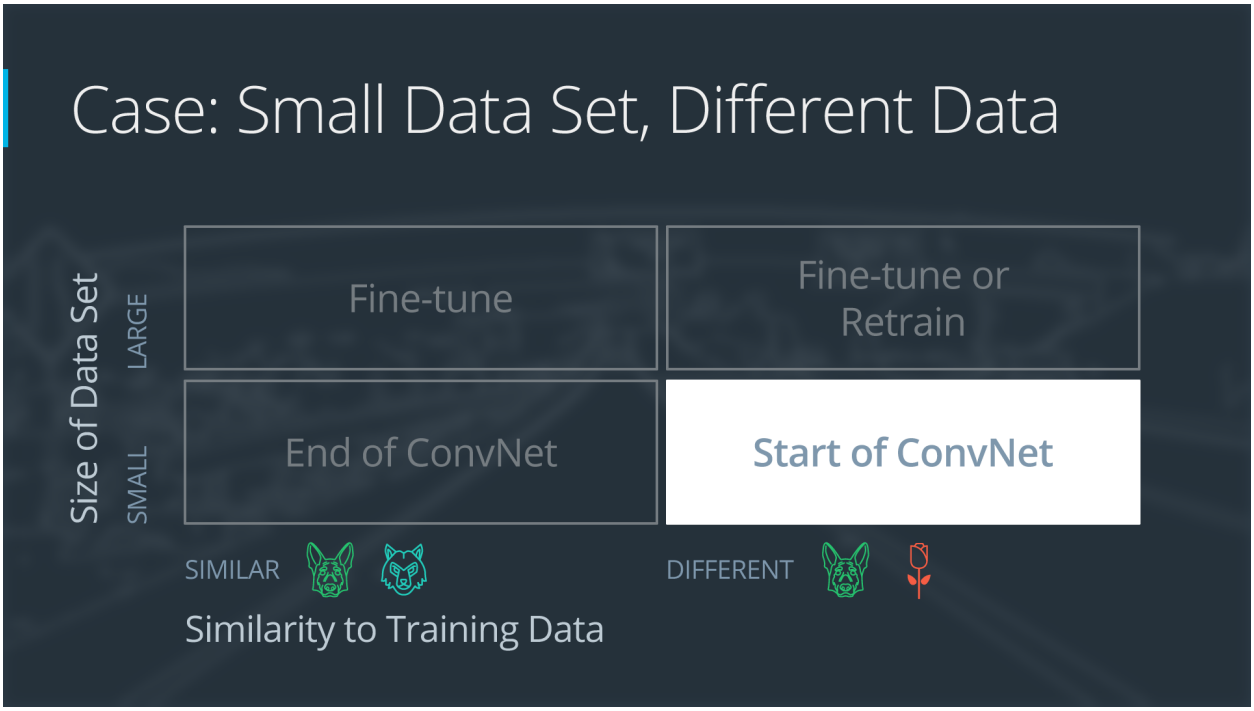

Figure 9: title

- train the network to update the weights of the new fully connected layer

Because the data set is small, overfitting is still a concern. To combat overfitting, the weights of the original neural network will be held constant, like in the first case.

But the original training set and the new data set do not share higher level features. In this case, the new network will only use the layers containing lower level features.

Here is how to visualize this approach:

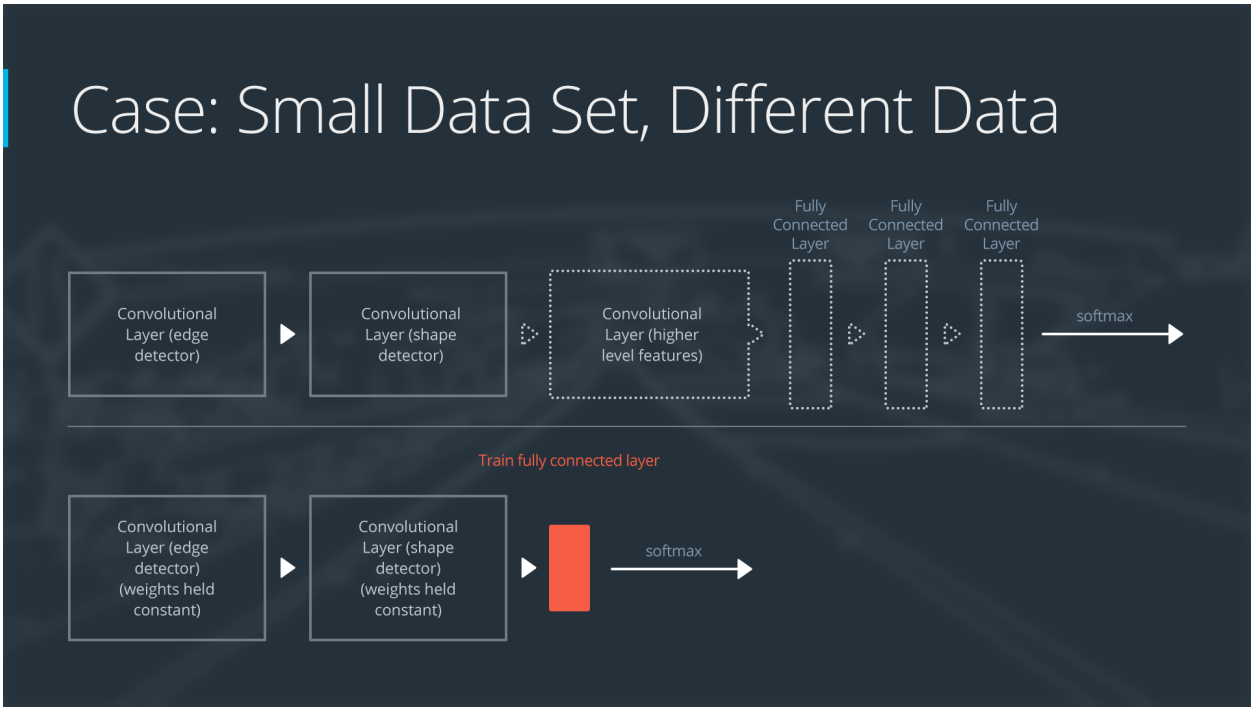

Figure 10: title

Case 3: Large Data Set, Similar Data If the new data set is large and similar to the original training data:

- remove the last fully connected layer and replace with a layer matching the number of classes in the new data set
- randomly initialize the weights in the new fully connected layer
- initialize the rest of the weights using the pre-trained weights
- re-train the entire neural network

Overfitting is not as much of a concern when training on a large data set; therefore, you can re-train all of the weights.

Because the original training set and the new data set share higher level features, the entire neural network is used as well.

Here is how to visualize this approach:

Case 4: Large Data Set, Different Data If the new data set is large and different from the original training data:

- remove the last fully connected layer and replace with a layer matching the number of classes in the new data set

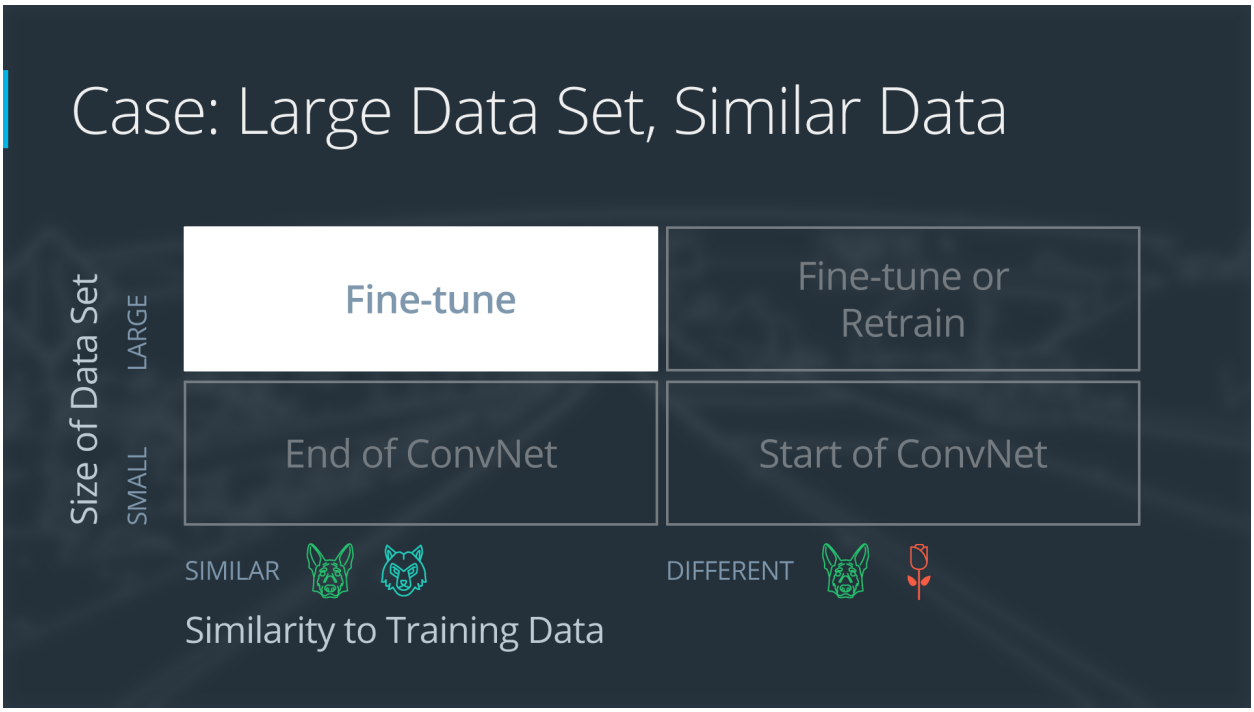

Figure 11: title

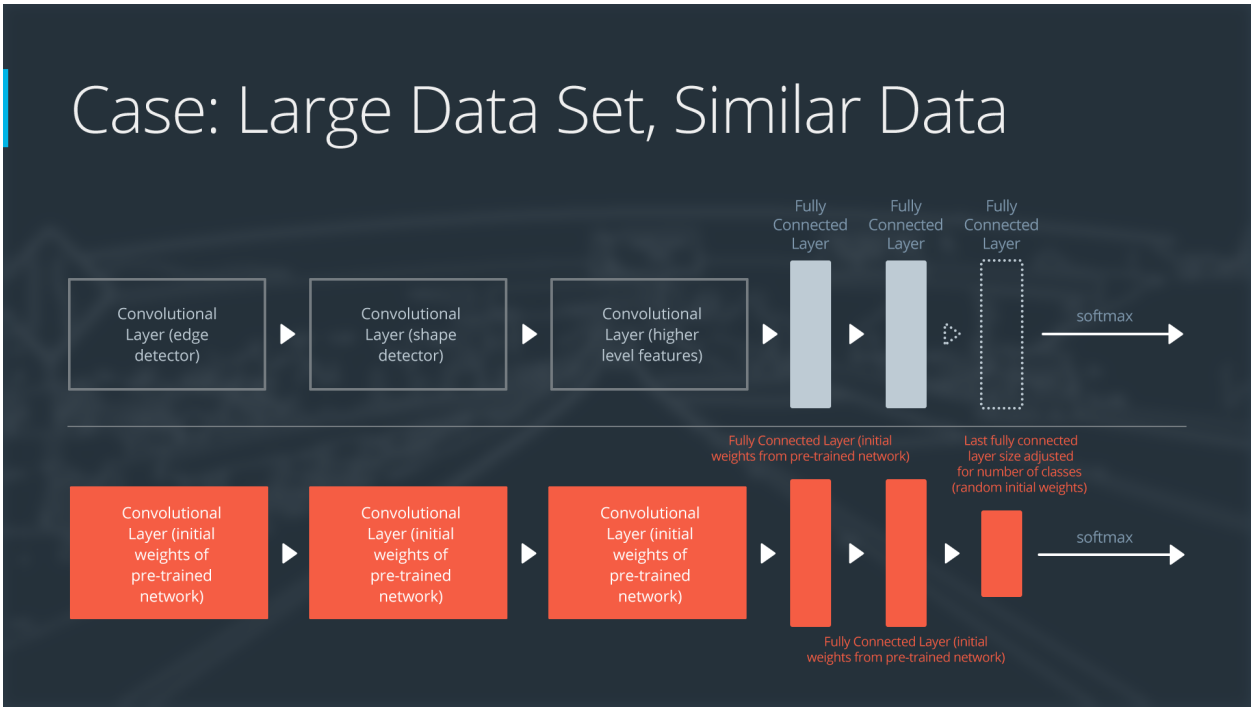

Figure 12: title

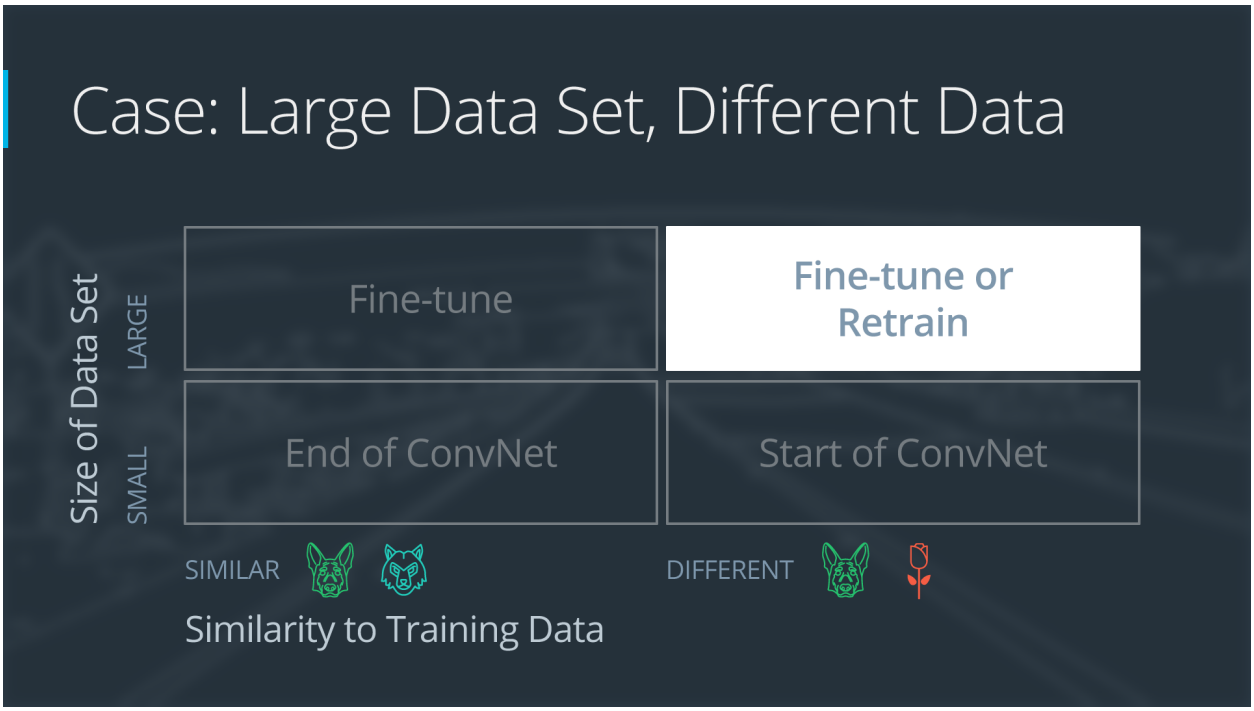

Figure 13: title

- retrain the network from scratch with randomly initialized weights
- alternatively, you could just use the same strategy as the "large and similar" data case

Even though the data set is different from the training data, initializing the weights from the pre-trained network might make training faster. So this case is exactly the same as the case with a large, similar data set.

If using the pre-trained network as a starting point does not produce a successful model, another option is to randomly initialize the convolutional neural network weights and train the network from scratch.

Here is how to visualize this approach:

Programming Transformer Neural Networks in PyTorch

Understanding Tokenization in NLP

Tokenization is a fundamental preprocessing step in Natural Language Processing (NLP) that involves breaking down text into smaller units called tokens. These tokens can be words, characters, subwords, or symbols, depending on the chosen tokenization method.

Basic Types of Tokenization

Word Tokenization

Word tokenization splits text into individual words. For example: "The cat sat on the mat" → ["The", "cat", "sat", "on", "the", "mat"]

This method faces challenges with:

- Contractions (don't, won't)

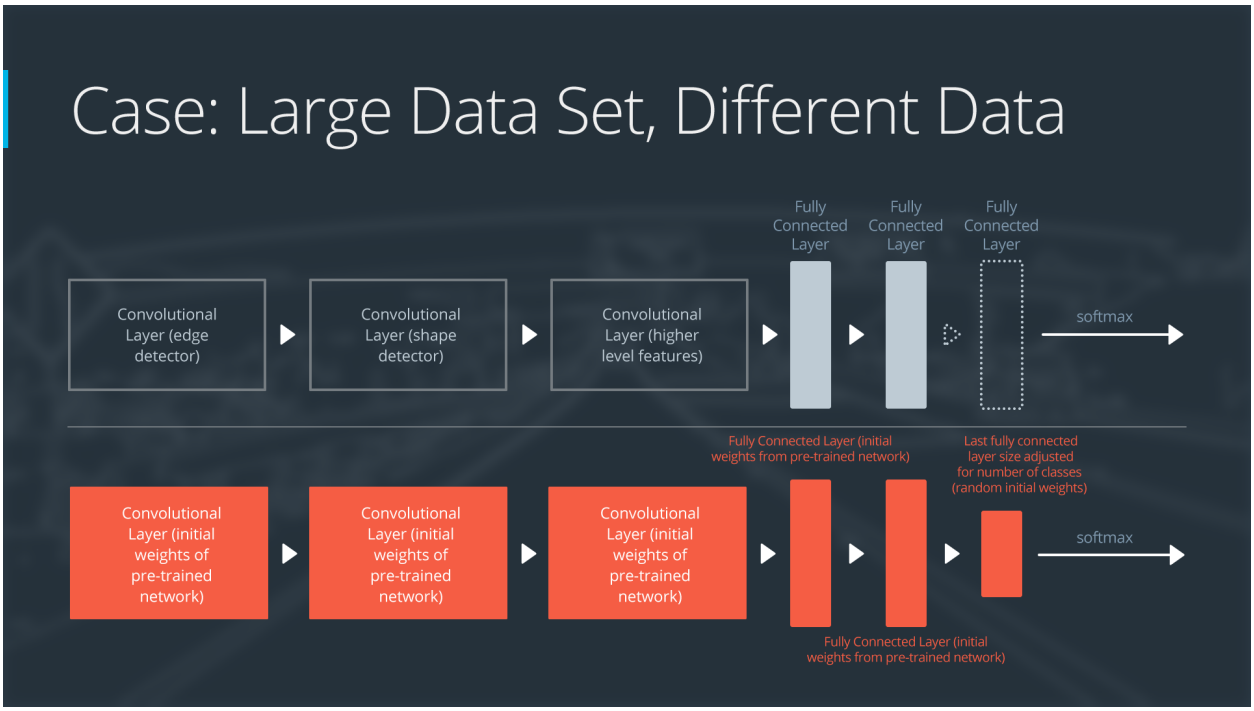

Figure 14: title

- Hyphenated words (state-of-the-art)
- Special characters and punctuation
- Numbers and dates

Character Tokenization

Character tokenization breaks text into individual characters, including spaces and punctuation. This approach is particularly useful for languages without clear word boundaries, like Chinese or Japanese.

Subword Tokenization

Modern NLP systems often use subword tokenization, which breaks words into meaningful subunits. Popular algorithms include:

- BPE (Byte Pair Encoding): Iteratively merges most frequent character pairs
- WordPiece: Similar to BPE but uses likelihood rather than frequency
- SentencePiece: Treats the text as a sequence of unicode characters
- Unigram: Uses a probabilistic model to find the optimal subword units

Advanced Considerations

Language-Specific Challenges

Different languages require different tokenization approaches:

- English: Dealing with contractions, compound words
- Chinese/Japanese: No spaces between words
- German: Long compound words
- Arabic: Complex morphology and diacritics

Special Token Handling

Modern tokenizers typically add special tokens for:

- Start of sequence [SOS]
- End of sequence [EOS]
- Padding [PAD]
- Unknown words [UNK]
- Masks for masked language modeling [MASK]

Impact on Model Performance

Tokenization choices significantly affect:

- Vocabulary size
- Model complexity
- Training efficiency
- Out-of-vocabulary word handling
- Overall model performance

Best Practices

- Choose tokenization method based on your specific task and language
- Consider vocabulary size implications
- Handle edge cases (URLs, emails, special characters)
- Maintain consistency between training and inference
- Document tokenization choices for reproducibility
- Test tokenization on representative data samples

Implementation Considerations

When implementing tokenization:

- Use established libraries (NLTK, spaCy, transformers)
- Consider preprocessing steps (lowercasing, removing special characters)
- Handle edge cases and errors gracefully
- Optimize for performance with large datasets
- Maintain consistent tokenization across your pipeline

Tokenization remains a critical step in NLP that can significantly impact model performance. Understanding its nuances and choosing the appropriate method for your specific use case is essential for successful NLP applications.

The Bag of Words Model How It Works The Bag of Words model uses a binary vector as an input. Each value in the vector is either one or zero, created from input token IDs.

Steps Involved

1. Tokenization:

- Start with an input sentence and split it into individual tokens.
- Convert all words to lowercase.
- Remove tokens for words that have very little useful information, such as “a,” “the,” and other stop words.
- Convert each token into a corresponding token ID.

2. Creating Input Vectors:

- For each token ID, if the ID value is, for example, 345, set one in the corresponding position in the vector.
- Repeat this process for all other input IDs.
- The input vector will have as many elements as there are unique words in the vocabulary. For instance, if the vocabulary has 65,000 words, the input vector will have 65,000 elements. Model Training and Application

With this binary vector, we can pass the input to a neural network or other machine learning models. Training a model with these input vectors allows it to perform tasks like classifying text (e.g., determining if messages are spam or non-spam). Limitations of the Bag of Words Model Ignores Word Order: The model does not consider the sequence in which words appear, which is crucial for understanding the context. Ignores Grammatical Structure: It overlooks punctuation and sentence structure, missing out on important linguistic nuances. Limited Semantic Meaning: The model's token representation only indicates the presence of words, lacking deeper semantic insights. Handling Unseen Words: The model struggles with words not seen during training, as they are not represented in the vocabulary. Moving Beyond Simple Models While the Bag of Words model can be effective for some simple tasks, it often falls short for more complex applications. To address these limitations, we will explore two key improvements:

Better Word Representation: Methods that provide more nuanced and semantically rich representations of words. Advanced Neural Network Architectures: More sophisticated models designed for NLP tasks, capable of capturing complex patterns and relationships in text. These advancements will help us build more robust and effective machine learning models for processing and understanding natural language.

Understanding Embedding Vectors in Machine Learning

Embedding vectors are dense, continuous numerical representations of discrete data in a lower-dimensional space. They capture semantic relationships and meaningful patterns in the data.

Core Concepts

Definition and Purpose

Embeddings convert categorical or discrete data into continuous vector representations where:

- Similar items are mapped to nearby points in the vector space
- Relationships between items are preserved through vector arithmetic
- Complex patterns can be captured in relatively few dimensions

Key Properties

- Dimensionality: Typically ranges from 50 to 1000 dimensions
- Density: All elements are meaningful, unlike sparse one-hot encodings
- Learned: Values are trained to optimize for specific tasks
- Transferable: Can be reused across different downstream tasks

Types of Embeddings

Word Embeddings

- Word2Vec: Uses skip-gram or CBOW architecture
- GloVe: Based on global word co-occurrence statistics
- FastText: Incorporates subword information
- BERT embeddings: Contextual representations

Other Common Applications

- Document embeddings

- Graph embeddings
- Product embeddings
- User embeddings
- Image embeddings

Training Methods

Supervised Learning

- Train embeddings directly for a specific task
- Use labeled data to guide the learning process
- Optimize for task-specific objectives

Self-Supervised Learning

- Predict context from input
- Masked language modeling
- Contrastive learning approaches
- Next sentence prediction

Transfer Learning

- Use pre-trained embeddings
- Fine-tune for specific tasks
- Leverage learned representations

Practical Applications

Natural Language Processing

- Semantic similarity
- Document classification
- Machine translation
- Question answering
- Named entity recognition

Recommender Systems

- User-item interactions
- Content-based filtering
- Collaborative filtering
- Cold-start problem solutions

Technical Considerations

Dimensionality

- Higher dimensions capture more information
- Too many dimensions lead to computational overhead
- Too few dimensions limit expressive power
- Need to balance accuracy vs. efficiency

Quality Assessment

- Intrinsic evaluation (similarity tasks)
- Extrinsic evaluation (downstream performance)
- Visualization techniques (t-SNE, PCA)

- Analogy testing

Implementation Best Practices

- Normalize vectors when appropriate
- Handle out-of-vocabulary items
- Consider computational resources
- Cache frequently used embeddings
- Use appropriate distance metrics

Advanced Topics

Multi-Modal Embeddings

- Combine different types of data
- Cross-modal retrieval
- Joint representation learning
- Alignment between modalities

Dynamic Embeddings

- Time-aware representations
- Adaptive embeddings
- Online learning approaches
- Contextual updates

Challenges and Solutions

- Handling rare items
- Scaling to large vocabularies
- Addressing bias in embeddings
- Maintaining interpretability
- Efficient storage and retrieval

Embedding vectors have revolutionized machine learning by providing powerful representations for discrete data. Their ability to capture semantic relationships and support transfer learning has made them essential in modern AI applications.

What Are Embedding Vectors?

Definition: Each word is represented as a smaller multi-dimensional vector, where each value represents certain characteristics of the word relevant to the model. Training: The size of the vectors is decided before training, and the model learns these vectors during the training process. Properties of Embedding Vectors

Interpretability: The specific values in embedding vectors are not directly interpretable, but they have meaningful properties. Dimensionality: Embedding vectors usually have hundreds or thousands of dimensions, making them challenging to visualize. Similarity in Embeddings

Semantic Similarity: Words with similar meanings are represented by similar vectors. For example, “laptop” and “computer” will have similar vector values and will be close to each other in the embedding space, while “orange” and “banana” will be close in their respective cluster. Computing Similarity: Dot Product

Dot Product Operation: Given two vectors, the dot product is calculated by multiplying corresponding values and summing the results. The result indicates the similarity:

Positive Result: Indicates similarity between words. Zero Result: Indicates unrelated words. Negative Result: Indicates opposite meanings. Example Calculations Semantic Relationships: Embedding vectors can reveal relationships: Gender Analogy: The difference between “woman” and “man” vectors, when added to “king,” approximates the “queen” vector. Geographical Analogy: The difference between “Rome” and

"Paris" vectors is similar to the difference between "France" and "Italy." Methods for Obtaining Word Embeddings Pre-trained Embedding Models: These are independent models trained specifically to compute word embeddings and can be used in various applications. Transformers and Embeddings: During the training of a Transformer model, word embeddings are learned as part of the process, eliminating the need for separate embedding libraries. Using Embedding Vectors in Models Application: Tokenize an input sentence and convert each token into an embedding vector. These vectors can then be flattened and passed into a neural network. Sequential Data Processing: Instead of using a feedforward neural network, architectures designed for sequential data, like text, are often more suitable.

How RNNs Work Sequential Data Processing Token-by-Token Processing: In RNNs, data is processed one token at a time. The first input token is passed through the network, and its output, along with the hidden state, is used as input for the next token. Hidden State: The hidden layer's output, called the hidden state, retains information about previous tokens, allowing the network to use this context when processing subsequent tokens. Initial State: An initial empty state is passed when processing the first token to start the sequence. Applications of RNNs RNNs are well-suited for tasks involving sequential data, such as:

Text Processing: Handling sequences in natural language processing. Audio Processing: Analyzing sequences in audio data. Time Series Analysis: Interpreting data points collected or recorded at time intervals. RNN Architectures Standard RNN One-to-One: Typically used for classification tasks, where only the final output is used to classify an input sequence, such as determining if a text is spam or not. Variations of RNNs One-to-Many RNN: Description: Takes an initial input and generates multiple output tokens. Use Cases: Tag generation, music generation. Many-to-Many RNN: Description: Processes an input sequence and generates an output sequence token by token. Use Cases: Language translation, sequence-to-sequence tasks. Challenges with RNNs Difficulty Accessing Previous States: RNNs struggle to retain information from tokens seen long ago due to the nature of the hidden state update process. Diminishing Gradient Problem: In deep networks, gradients can become very small, slowing down training and making it difficult to capture dependencies over long sequences. Solutions and Improvements LSTM (Long Short-Term Memory): Designed to better retain information over longer sequences by using gates to control the flow of information. GRU (Gated Recurrent Unit): Simplifies the LSTM model while still addressing some of its key challenges. The Next Evolution: Transformers The next major advancement in NLP came with the Transformer architecture, which uses the attention mechanism to handle the challenges RNNs face, especially with long-range dependencies and gradient issues.

By understanding RNNs and their variations, we gain insight into why newer architectures like Transformers have been so revolutionary in the field of NLP. In the following sections, we will explore these advanced architectures and their benefits.

1. Recurrent Neural Networks (RNN)

RNNs are neural networks designed to work with sequential data by maintaining an internal state (memory) that gets updated as they process input sequences.

Core Architecture

- Sequential processing of inputs
- Hidden state maintenance
- Recurrent connections
- Ability to handle variable-length sequences

Types of RNNs

1. Simple RNN (Vanilla)
    - Basic recurrent structure
    - Suffers from vanishing/exploding gradients
2. LSTM (Long Short-Term Memory)

- Gates: Input, Forget, Output
- Cell state for long-term memory
- Better gradient flow
- More stable training

3. GRU (Gated Recurrent Unit)
   - Simplified version of LSTM
   - Reset and Update gates
   - Fewer parameters than LSTM
   - Often similar performance

Common Applications

- Natural Language Processing
- Time Series Prediction
- Speech Recognition
- Music Generation
- Video Processing

Limitations

- Sequential processing is slow
- Limited parallel processing
- Difficulty with long-range dependencies
- Vanishing/exploding gradients
- Memory constraints

2. Transformer Architecture

Transformers revolutionized deep learning by introducing self-attention mechanisms and parallel processing capabilities.

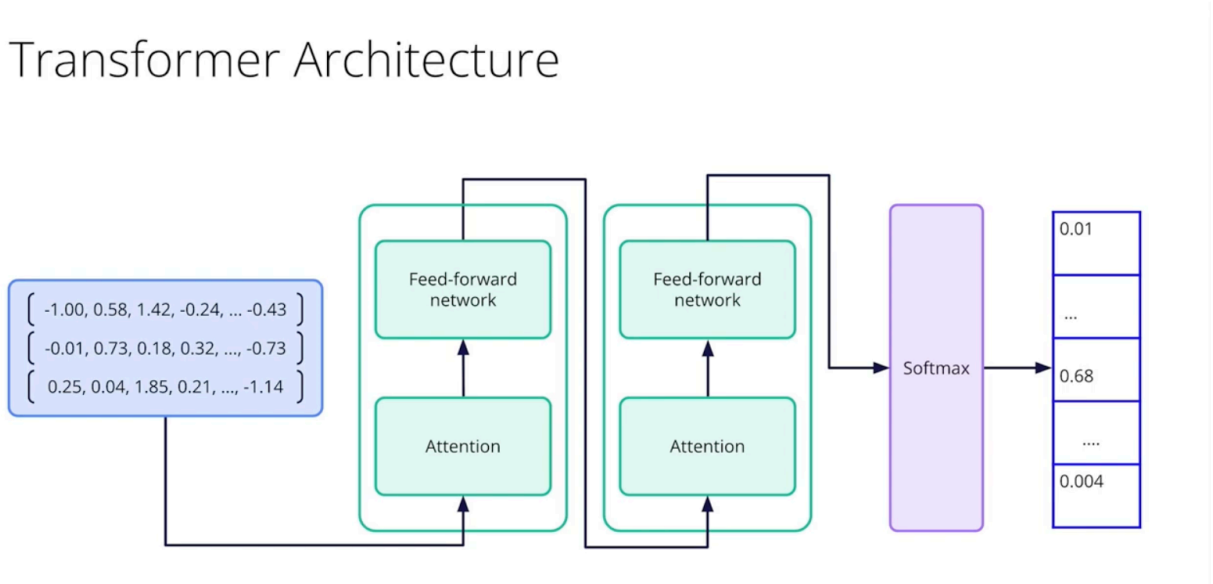

Figure 15: title

Core Components

Encoder

- Self-attention layers
- Feed-forward networks
- Layer normalization
- Residual connections

Decoder

- Masked self-attention
- Encoder-decoder attention
- Feed-forward networks
- Output generation

Key Mechanisms

Self-Attention

- Query, Key, Value matrices
- Attention scores computation
- Scaled dot-product attention
- Multi-head attention

Positional Encoding

- Adds position information
- Sine/cosine functions
- Learnable positions
- Maintains sequence order

Advantages

- Parallel processing
- Better long-range dependencies
- No recurrence needed
- Stable training
- State-of-the-art performance

Applications

- Machine Translation
- Text Generation
- Document Summarization
- Image Recognition
- Speech Processing
- Protein Structure Prediction

Architecture Variants

1. BERT
   - Bidirectional encoder
   - Masked language modeling
   - Next sentence prediction
2. GPT

- Decoder-only architecture
- Autoregressive modeling
- Large-scale training

3. T5
   - Text-to-text framework
   - Unified approach
   - Transfer learning

Implementation Considerations

Training

- Large computational requirements
- Extensive data needed
- Careful hyperparameter tuning
- Memory management
- Gradient accumulation

Optimization

- Model parallelism
- Mixed precision training
- Efficient attention mechanisms
- Knowledge distillation
- Model pruning

Both architectures have their place in modern deep learning:

- RNNs excel at smaller sequential tasks with clear temporal dependencies
- Transformers dominate large-scale language tasks and complex sequence modeling

The choice between them depends on:

- Task requirements
- Available compute resources
- Data characteristics
- Latency requirements
- Model interpretability needs

Overview of Transformer Model Functionality Tokenization and Embeddings: The input prompt is split into tokens, each converted into a token ID and then into an embedding vector. The same token ID consistently produces the same embedding vector. Attention Mechanism: Embedding vectors are refined in the attention block. Example: The word "date" in different sentences can have different meanings. The attention mechanism updates the embedding vector based on the context provided by other words in the input. The process of interaction between embedding vectors is called attention, which uses attention scores to show the relationships between input tokens. Transformer Block: Comprises an attention block followed by a feedforward network. Multiple Transformer blocks (layers) are chained together, with the output from one layer becoming the input for the next. Output and Softmax: The final layer provides a probability distribution for the next token to generate, using the softmax function. Benefits of Transformers Parallel Processing: Transformers can process the entire input prompt simultaneously, unlike RNNs. Efficiency with Long Texts: Better handling of longer texts. Ease of Training: More straightforward training compared to RNN models. Performance: Superior performance on real-world tasks. Concept of Context Window Context Window: The maximum number of tokens a model can handle at once, including both the input prompt and the output generated so far. Variability: The size of the context window varies by model: Smaller models may handle around 2,000 tokens. Larger models, like recent Gemini models, can handle over 1.5 million tokens. Tokenization: A single word may be split into multiple tokens. For example, in GPT models, it typically takes four tokens to represent three words.

Building the Transformer Neural Network | PyTorch

Model Architecture

- Similarity to GPT Models: Our model will have a similar architecture to the GPT models created by OpenAI. Differences from GPT Models:
- Model Size: Our model will be much smaller, with significantly fewer parameters, making it more accessible and less resource-intensive to train.
- Tokenizer Type: We will use a character-level tokenizer instead of a subword tokenizer to simplify our implementation.
- Focus on Understandability: The model will prioritize clarity and educational value over performance.

Implementation Details

1. Model Configuration: Once the implementation is complete, we will be able to create an instance of the model and set various parameters, such as context window size.
2. Text Generation: Using the trained model and a provided prompt, we can generate text that mimics Shakespeare's style, such as lines that could be spoken by characters like Romeo.

Key Steps in Data Preparation

1. Implementing a Tokenizer Purpose: A tokenizer converts text into token IDs, which are numerical representations of the text. We'll implement a tokenizer in PyTorch
2. Preparing the Training Dataset Goal: The model's goal is to predict the next token given part of an input dataset. Example: Given the phrase: "The core innovation of Transformers is the attention." Input: "The" → Model predicts: "core" Input: "The core" → Model predicts: "innovation" Continue until the last input: "The core innovation of Transformers is the" → Model predicts: "attention" Dataset Representation: Inputs and targets are part of the same phrase but are shifted by one position. While the model sees token IDs during training, for clarity, we'll often refer to these as words. Training Process Training Batches: During each training iteration, we don't just use a single example. Instead, we use multiple random training examples to create a training batch. Efficiency: Using batches helps better utilize GPU resources, speeding up the training process.

Using DataLoader for Mini-Batches In training a model, we typically don't use a single sample per training iteration, as this approach (stochastic gradient descent) doesn't efficiently utilize resources and leads to fluctuating training losses. Using the whole dataset (batch gradient descent) is impractical due to high computational costs and memory requirements.

Mini-Batches Instead, models are often trained using mini-batches, where each training iteration uses a small subset of samples. This approach, known as mini-batch gradient descent, balances efficiency and stability in training.

Embeddings and Positional Layers in a Transformer-Based Language Model

Building the Transformer Model Step 1: Processing Input Tokenization: The input text is tokenized, converting each word into a token ID. Word Embeddings: Each token ID is then converted into a corresponding word embedding. After this step, we have a matrix of embedding vectors, each representing a token in the input. Key Points About Word Embeddings Dependence on Token IDs: The word embeddings are solely dependent on the token IDs; the same token ID always produces the same word embedding vector. Training: These embeddings are not pre-defined but are learned during the training of our model. Creating Embeddings Vocabulary Size: Set to 10,000 (arbitrary choice). Embedding Vector Size: Set to 75. nn.Embedding: A PyTorch class used to create embeddings. It requires the vocabulary size and embedding vector size as inputs. Usage Token IDs: We obtain token IDs using a tokenizer, but for now, we use arbitrary IDs for illustration. Embedding Layer: Pass these token IDs to the nn.Embedding instance, which returns a tensor with vectors of size 75 (the embedding size). Adding Positional Encoding Before feeding the embedding vectors into the model, we add positional encoding:

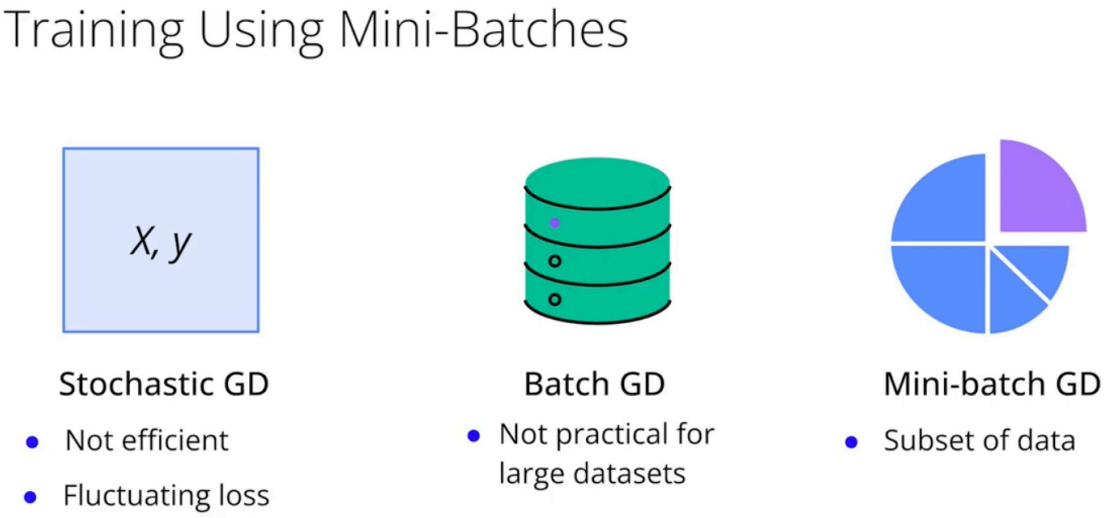

Figure 16: title

title

Figure 17: title

Purpose: Injects positional information into the word embeddings, helping the model understand the relative order of input tokens. Positional Encoding Matrix: A fixed matrix added to the word embeddings. This matrix does not depend on the input and provides consistent positional information. Positional Encoding in Models Training: In some models, including ours, the values in the positional encoding matrix are learned along with other model parameters. Fixed Formula: In other large language models (LLMs), positional encodings are defined using a fixed formula and remain unchanged during training.

Building a Transformer Model

Input Processing and Embeddings

Token Processing

The Transformer begins by converting input text into meaningful numerical representations through two key steps:

1. Initial Tokenization
   - Raw text is converted to token IDs
   - Each token gets a unique numerical identifier
   - Example: [348, 1978, 634] as shown in the image
2. Word Embeddings Generation
   - Token IDs are transformed into dense vector representations
   - Creates rich semantic representations
   - Example vectors: [-1.00, 0.58, 1.42, -0.24, ..., -0.43]

Positional Encoding

Purpose and Implementation

Positional encoding solves the critical problem of sequence order in Transformers:

- Adds position-dependent information to each token
- Generates unique patterns for each position
- Combines with word embeddings through addition
- Example vectors: [0.77, 0.15, 0.63, -1.36, ..., 0.06]

Characteristics

- Position-specific patterns
- Maintains consistent spacing between positions
- Scales with sequence length
- Learned or fixed depending on implementation

Final Input Representation

The final input to the Transformer model combines:

- Word embeddings (semantic meaning)
- Positional encodings (sequence information)

As shown in the image, these are combined through element-wise addition before being fed into the Transformer model proper.

Model Architecture Considerations

- Embedding dimension consistency across all components
- Careful initialization of both embedding types
- Proper scaling to prevent dominance of either component
- Integration with subsequent Transformer layers

This forms the foundation for the Transformer's ability to process sequential data while maintaining awareness of both meaning and position.

Understanding the Softmax Function

Definition and Purpose

The softmax function, also known as the normalized exponential function, transforms a vector of K real numbers into a probability distribution of K possible outcomes.

Mathematical Definition

For a vector z = (z , ..., z), the softmax function (z) is defined as:

 (z) = exp(z) / Σ exp(z)

where:

- exp(x) is the exponential function
- i = 1, ..., K
- j ranges over 1 to K in the sum

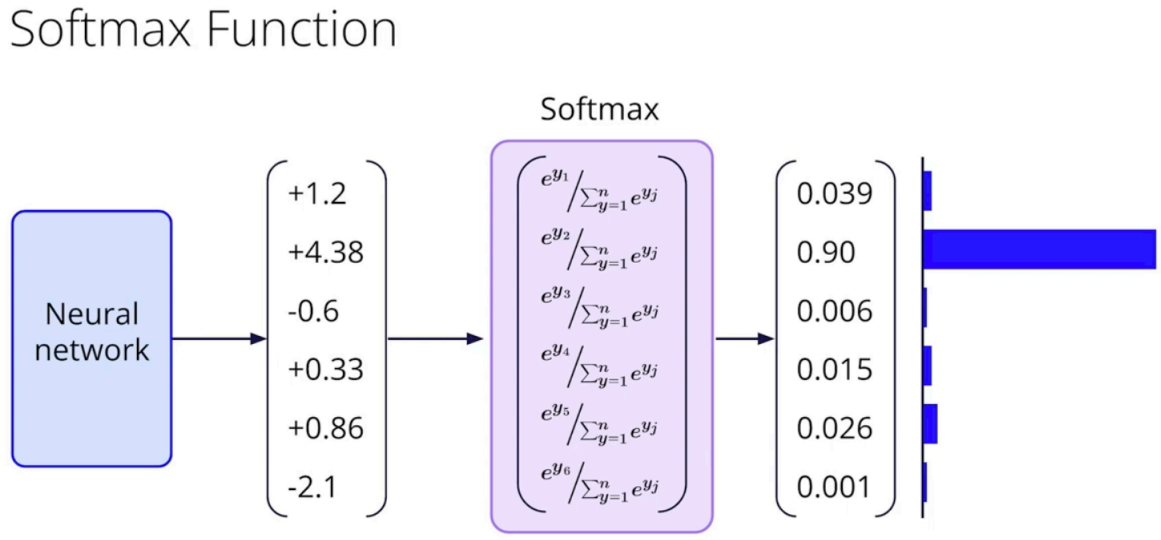

Figure 18: title

Key Properties

Probability Distribution

- Output values sum to 1: Σ (z) = 1
- Each output is in range (0,1): 0 < (z) < 1

Mathematical Characteristics

- Differentiable: Enables gradient-based optimization
- Monotonic: Preserves relative ordering of inputs
- Scale invariant: (z + c) = (z) for any constant c

Derivative Formula

The derivative of softmax with respect to its inputs is:

(z) / z = (z) (- (z))

where:

- is the Kronecker delta
- i,j are indices of the vector elements

Numerical Stability

To prevent overflow/underflow, implementation often uses:

log_softmax(z) = z - log(Σ exp(z))

Common stabilization technique:

```
z_max = max(z)
exp_z = exp(z - z_max)
softmax = exp_z / sum(exp_z)
```

Applications

1. Classification Tasks
   - Output layer of neural networks
   - Multi-class probability distribution
   - Cross-entropy loss computation
2. Attention Mechanisms
   - Computing attention weights
   - Normalizing similarity scores
3. Reinforcement Learning
   - Policy networks
   - Action probability distribution

Implementation Considerations

Performance Optimization

- Vectorized operations
- GPU acceleration
- Memory efficiency
- Numerical precision

Common Pitfalls

- Overflow/underflow in naive implementations
- Gradient vanishing with extreme inputs
- Computational cost with large dimensions

The softmax function remains fundamental in machine learning, particularly in:

- Neural networks
- Natural language processing
- Computer vision
- Probabilistic modeling

Understanding its properties and implementation details is crucial for effective model development.

Attention Mechasnism The attention mechanism in Transformers works like a smart note-taking system during a group conversation. Imagine you're in a meeting and need to focus on different speakers at different times - sometimes you pay more attention to one person because what they're saying is more relevant to your current topic, while other times you need to connect information from multiple speakers. The attention mechanism does exactly this with words or tokens in a sequence - it helps the model figure out which other words are most important for understanding the current word, allowing it to create rich, context-aware representations of language.

The attention mechanism computes weighted relationships between all elements in a sequence using Query (Q), Key (K), and Value (V) matrices, derived from the input embeddings through learned linear transformations. The core computation is Attention(Q,K,V) = softmax(QK^T/√d_k)V, where d_k is the dimension of the keys. This is typically implemented as Multi-Head Attention, where the attention computation is performed in parallel across several heads, each learning different relationship patterns. The mechanism employs scaled dot-product attention, where the scaling factor ($1/\sqrt{d\_k}$) prevents the softmax function from entering regions with extremely small gradients. The resulting attention weights form a probability distribution through softmax, which is then used to create a weighted sum of the values, producing context-aware representations for each position in the sequence.

Calculating Attention Overview of the Attention Block Attention Block Function: Computes attention scores and updates embedding values using these scores. Components: The attention block involves several

steps, including calculating query, key, and value matrices and updating embedding vectors. Step-by-Step Implementation Query Matrix (Q) Calculation: Multiply the matrix of embedding vectors by another matrix (W_Q) to produce the query matrix (Q). Notation: Represented as Q for simplicity, often visualized as an array of vectors. Key Matrix (K) Calculation: Similar to Q, the key matrix (K) is obtained by multiplying the embedding vectors by matrix (W_K). Concept: Queries can be seen as questions about other words, while keys provide answers. Computing Attention Scores Similarity Check: Calculate the dot product between vectors in the Q and K matrices to determine the similarity. Matrix Multiplication: Multiply the Q matrix by the transposed K matrix to get the attention scores. Normalization with Softmax Purpose: Ensures all values are between 0 and 1 and sum to one. Application: Apply the softmax function to each column of the resulting matrix. Diagram of Operations Steps Recap: Derive Q and K matrices from the input embeddings. Compute attention scores via matrix multiplication and softmax. Use these scores to update embedding vectors. Value Matrix (V) Derivation: Similar to Q and K, the value matrix (V) is derived using another matrix (W_V). Updating Embeddings: Multiply each vector in V by the corresponding attention scores to produce updated embeddings. Final Diagram of the Attention Block Complete Process: Compute Q, K, and V matrices. Compute attention scores from Q and K. Update embeddings using attention scores and V. Multi-Head Attention Purpose: Multiple attention heads allow the model to learn different relationships between words in a sentence. Process: Each head performs similar calculations with different parameters, and the results are concatenated to form new embedding vectors. Formula for Attention From "Attention Is All You Need" Paper: Components: Uses Q, K, and V matrices. Adjustment: Includes a division by the square root of the dimension of the K matrix vector to prevent large attention scores. Final Steps: Apply softmax and multiply by V to compute the final output. Understanding the attention mechanism is key to grasping how Transformer models process and generate text, enabling them to focus on relevant parts of the input sequence when making predictions.

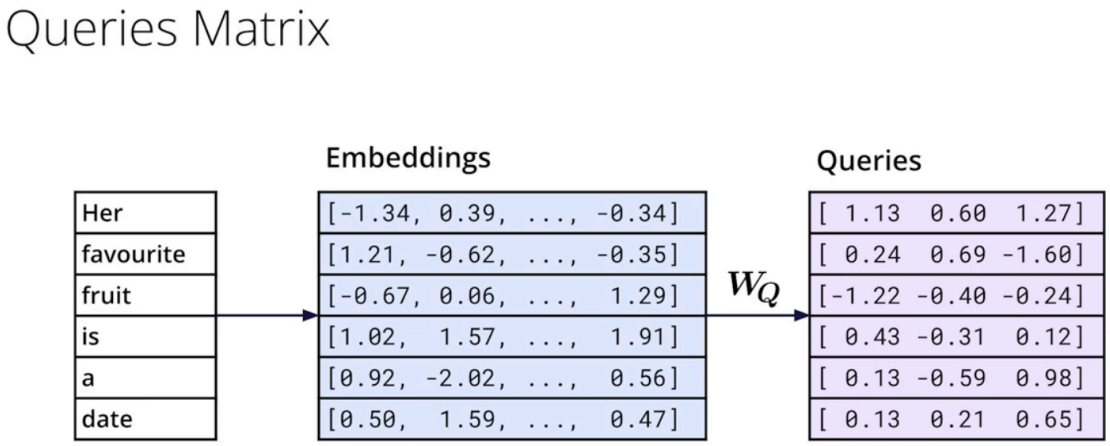

Figure 19: title

Attention Mechanism in Transformers

Core Concepts

Self-Attention Formula

The fundamental attention calculation is:

Attention(Q, K, V) = softmax(QK^T/√d_k)V

Queries and Keys

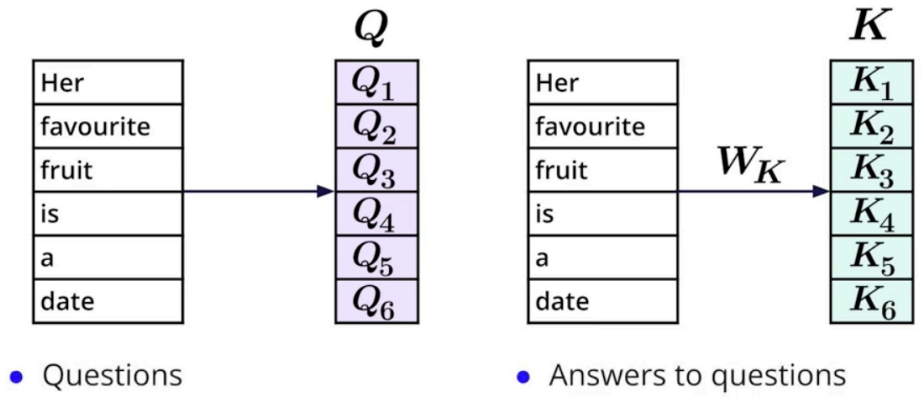

Figure 20: title

Calculating Attention Scores

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6
K_1	$K_1 \cdot Q_1$	$K_1 \cdot Q_2$	$K_1 \cdot Q_3$	$K_1 \cdot Q_4$	$K_1 \cdot Q_5$	$K_1 \cdot Q_6$
K_2	$K_2 \cdot Q_1$	$K_2 \cdot Q_2$	$K_2 \cdot Q_3$	$K_2 \cdot Q_4$	$K_2 \cdot Q_5$	$K_2 \cdot Q_6$
K_3	$K_3 \cdot Q_1$	$K_3 \cdot Q_2$	$K_3 \cdot Q_3$	$K_3 \cdot Q_4$	$K_3 \cdot Q_5$	$K_3 \cdot Q_6$
K_4	$K_4 \cdot Q_1$	$K_4 \cdot Q_2$	$K_4 \cdot Q_3$	$K_4 \cdot Q_4$	$K_4 \cdot Q_5$	$K_4 \cdot Q_6$
K_5	$K_5 \cdot Q_1$	$K_5 \cdot Q_2$	$K_5 \cdot Q_3$	$K_5 \cdot Q_4$	$K_5 \cdot Q_5$	$K_5 \cdot Q_6$
K_6	$K_6 \cdot Q_1$	$K_6 \cdot Q_2$	$K_6 \cdot Q_3$	$K_6 \cdot Q_4$	$K_6 \cdot Q_5$	$K_6 \cdot Q_6$

Softmax | Softmax | Softmax | Softmax | Softmax | Softmax

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6
K_1	0.731	0.046	0.057	0.056	0.045	0.063
K_2	0.04	0.827	0.025	0.027	0.034	0.044
K_3	0.017	0.026	0.841	0.04	0.033	0.04
K_4	0.025	0.03	0.044	0.839	0.03	0.029
K_5	0.089	0.065	0.039	0.063	0.7	0.037
K_6	0.05	0.066	0.036	0.034	0.046	0.765

Figure 21: title

Using Attention Scores

		Q_1	Q_2	Q_3	Q_4	Q_5	Q_6
V_1	K_1	0.731	0.046	0.057	0.056	0.045	0.063
V_2	K_2	0.04	0.827	0.025	0.027	0.034	0.044
V_3	K_3	0.017	0.026	0.841	0.04	0.033	0.04
V_4	K_4	0.025	0.03	0.044	0.839	0.03	0.029
V_5	K_5	0.089	0.065	0.039	0.063	0.7	0.037
V_6	K_6	0.05	0.066	0.036	0.034	0.046	0.765

$$0.046 \bullet V_1 + 0.827 \bullet V_2 + 0.026 \bullet V_3 + 0.03 \bullet V_4 + 0.065 \bullet V_5 + 0.066 \bullet V_6$$

[1.86, 1.21, -0.02, 0.30, ..., 0.96]

Figure 22: title

Attention

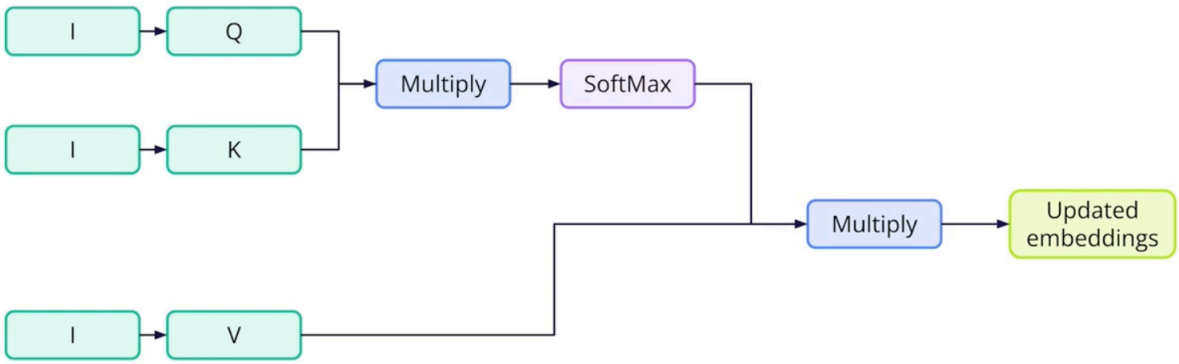

Figure 23: title

where:

- Q = Query matrix
- K = Key matrix
- V = Value matrix
- d_k = Dimension of keys
- √d_k = Scaling factor

Components and Process

1. Query, Key, Value Generation

Each input embedding is transformed into:

- Query: What the current position is looking for
- Key: What the position offers to others
- Value: The actual content to be aggregated

2. Attention Weights

- Compute similarity scores between Q and K
- Apply scaling factor ($1/\sqrt{d\_k}$)
- Pass through softmax for normalization
- Results in attention probability distribution

3. Multi-Head Attention

- Splits attention into multiple parallel heads
- Each head learns different aspects of relationships
- Formula: MultiHead(Q,K,V) = Concat(head , ..., head)W^O
- Each head = Attention(QW Q, KW K, VW V)

Types of Attention

1. Self-Attention

- Relates different positions within same sequence
- Q, K, V all come from same source
- Used in both encoder and decoder

2. Cross-Attention

- Relates positions between encoder and decoder
- Q from decoder, K and V from encoder
- Enables sequence-to-sequence learning

3. Masked Attention

- Used in decoder self-attention
- Prevents attending to future positions
- Implements autoregressive property

Implementation Details

Efficiency Considerations

- Parallel computation
- Memory requirements: $O(n^2)$ where n is sequence length

- Attention matrix sparsification techniques
- Optimized implementations for long sequences

Key Advantages

- Captures long-range dependencies
- No sequential processing requirement
- Bidirectional context integration
- Interpretable attention weights

Advanced Concepts

Variations and Improvements

- Relative Positional Encoding
- Sparse Attention Patterns
- Linear Attention
- Local-Global Attention

Performance Optimizations

- Efficient attention implementations
- Memory-efficient techniques
- Gradient accumulation
- Mixed precision training

Practical Applications

Common Use Cases

- Machine Translation
- Document Understanding
- Speech Recognition
- Image Processing
- Cross-Modal Tasks

Implementation Guidelines

- Proper initialization
- Layer normalization placement
- Dropout strategies
- Residual connections

The attention mechanism revolutionized deep learning by enabling direct modeling of dependencies regardless of their distance in the sequence, making it a cornerstone of modern transformer architectures.

Masking Preventing Cheating in Transformer Training with Masked Multi-Head Attention When training Transformer models, we convert parts of the training dataset into multiple training examples.

However, a problem arises because the model can "cheat" by looking ahead at the entire input when predicting the next word.

To prevent this, we need to modify the attention scores to ensure the model cannot access future tokens.

Solution: Masked Multi-Head Attention To stop the model from "cheating," we implement a masking strategy:

1. Attention Matrix Adjustment:

Masking in the Attention Matrix

	Q_1	Q_2	Q_3	Q_4	Q_5
K_1	$K_1 \cdot Q_1$	$K_1 \cdot Q_2$	$K_1 \cdot Q_3$	$K_1 \cdot Q_4$	$K_1 \cdot Q_5$
K_2	$-\infty$	$K_2 \cdot Q_2$	$K_2 \cdot Q_3$	$K_2 \cdot Q_4$	$K_2 \cdot Q_5$
K_3	$-\infty$	$-\infty$	$K_3 \cdot Q_3$	$K_3 \cdot Q_4$	$K_3 \cdot Q_5$
K_4	$-\infty$	$-\infty$	$-\infty$	$K_4 \cdot Q_4$	$K_4 \cdot Q_5$
K_5	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$K_5 \cdot Q_5$

Softmax | Softmax | Softmax | Softmax | Softmax

	Q_1	Q_2	Q_3	Q_4	Q_5
K_1	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$
K_2	0	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$
K_3	0	0	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$
K_4	0	0	0	$P_{4,4}$	$P_{4,5}$
K_5	0	0	0	0	$P_{5,5}$

Figure 24: title

Masked Multi-Head Attention

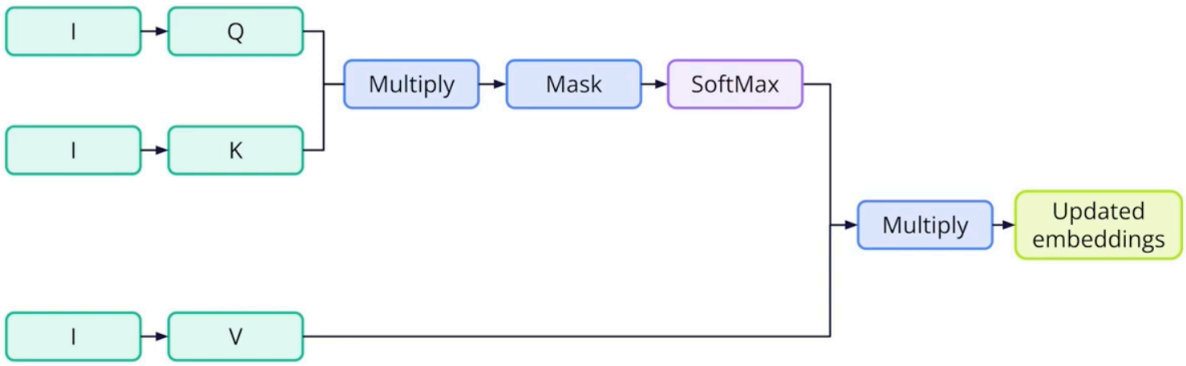

Figure 25: title

- Problem: The model can see the next token while predicting it.
- Solution: Update the attention matrix to mask out certain values.

2. Masking Strategy:

- Method: Assign all values below a certain diagonal to negative infinity.
- Softmax Effect: The softmax function converts these negative infinity values to zeros, effectively ignoring them.

3. Implementation:

- Diagram Update: In our multi-head attention diagram, we add the masking step after matrix multiplication and before applying softmax.
- Masked Multi-Head Attention: This approach is widely used in Transformer models, including GPT models.

By applying a triangular mask, we ensure that the model only considers current and previous tokens when making predictions, thus maintaining the integrity of the training process.

Dropout Layer

What is a Dropout Layer? Function in Neural Networks: In a feedforward neural network, each output from a previous layer is connected to all neurons in the next layer. Purpose of Dropout: The Dropout layer introduces randomness by setting a certain percentage of the inputs to zero during training. This percentage is usually specified as a hyperparameter. How Dropout Works Random Selection: On each training iteration, different inputs are randomly selected and set to zero. This list of " disabled" inputs changes with each iteration. Generalization: By doing this, the model does not become overly reliant on specific features, which helps it generalize better to new, unseen data. This process reduces the risk of overfitting. Training vs. Inference During Training: The Dropout layer is active, randomly disabling inputs to help generalization. During Inference: When the model is used for generating text or making predictions, the Dropout layer is disabled, ensuring that all features are fully utilized. Implementation in PyTorch Prebuilt in PyTorch: We do not need to implement the Dropout layer ourselves, as PyTorch provides a built-in implementation through the Dropout class.

The Dropout technique is a crucial component in training robust neural networks, especially when dealing with complex data and models. It ensures that the model does not overfit and can generalize well across different datasets.

Implementing the Attention Block

Model Configuration Before we start, we need to define a configuration for our model, containing hyperparameters that define its structure. This configuration is stored in a dictionary with the following parameters:

Vocabulary Size: The number of unique token IDs supported by the tokenizer. Context Size: The maximum number of tokens the model can see at once. Embedding Dimension: The size of the embedding vectors. Number of Heads: The number of attention heads, each processing input independently. Number of Layers: The number of Transformer blocks in the model. Dropout Rate: The proportion of outputs set to zero in Dropout layers. Use Bias: A boolean indicating whether the linear transformations should include bias terms.

Implementing the Transformers Block Overview of a Transformer Block

1. Input Processing: Each input token is converted into an embedding vector and passed to the multi-head attention block.
2. Feedforward Network: The output from the multi-head attention block is sent to a feedforward network layer.3. 3. Chaining: A Transformer model consists of multiple such blocks chained together.

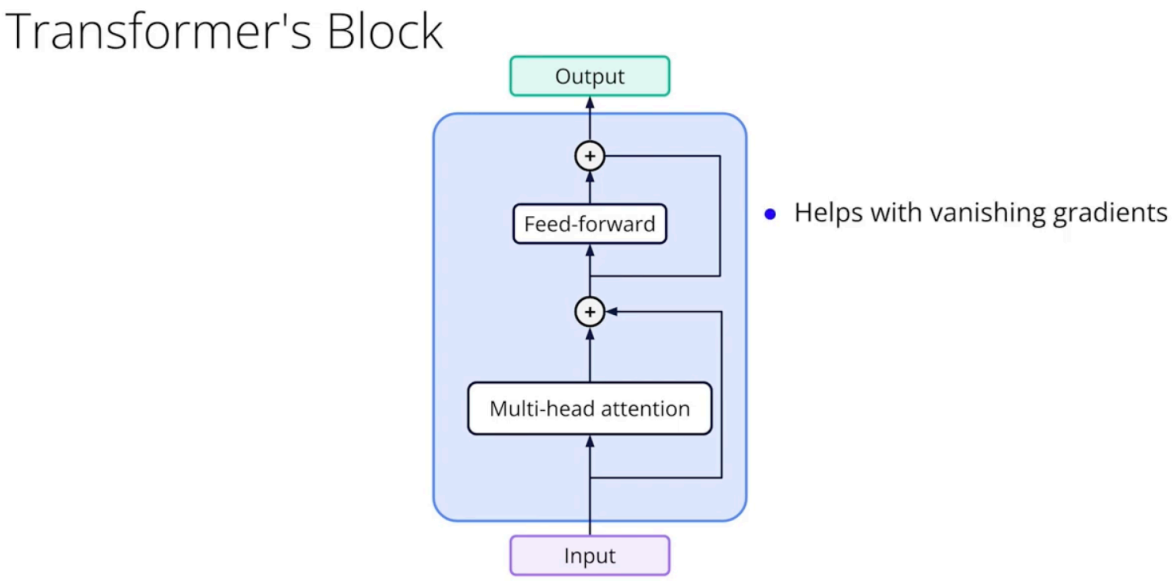

Figure 26: title

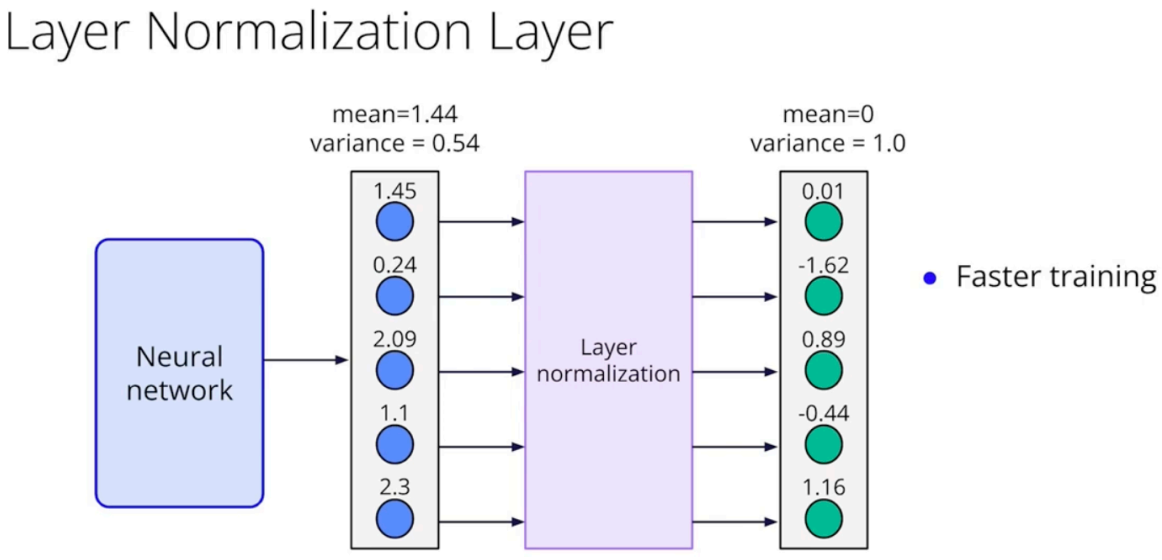

Figure 27: title

Feedforward Layer

1. Function: Applied to each embedding vector individually.
2. Process:

- Linear Layer: Each embedding vector is passed to a linear layer with an output size four times larger than the input size.
- Activation Function: The GELU activation function is applied to introduce nonlinearity.
- Size Reduction: The output vector is then reduced to the original embedding size.

GELU Activation Function Comparison: GELU is similar to the ReLU activation function but has a smoother shape, which helps in achieving better training results. Additional Components

Residual Connections Purpose: Helps with the vanishing gradient problem in deep networks. Implementation: The input to a block is added to the block's output before passing it forward, improving training efficiency.

Layer Normalization and Dropout Layer Normalization: Scales the output so that the mean becomes zero and variance becomes one, leading to faster training. Dropout Layer: Helps prevent overfitting by setting a portion of the output to zero during training. Summary Components of a Transformer Block: Multi-Head Attention Feedforward Network Layer Normalization Dropout Residual Connections These elements work together to process and refine the input data, making Transformer models highly effective for various NLP tasks.

Feedforward Block Implementation To define a feedforward block, we create a new class inheriting from nn.Module.

Key Components Linear Layers: The first linear layer projects the input dimension to four times its size, followed by a GELU activation function and another linear layer that reduces the dimension back to the original size. Dropout Layer: Added to prevent overfitting.

Transformer Block Implementation With both the multi-head attention and feedforward blocks ready, we can now implement the Transformer block.

Transformer Block Components

- Multi-Head Attention: Uses the multi-head attention block.
- Feedforward Block: Incorporates the previously defined feedforward block.
- Layer Normalization: Two normalization layers are included to stabilize the training process.
- Residual Connections: Two residual connections are used to help with the vanishing gradient problem.

Assembling the Transformer Model

Now that we have implemented a single block, let's look at how to assemble everything into a complete Transformer model. We'll walk through all the components of the model before implementing it in PyTorch.

Model Components Overview

1. Tokenization and Embedding:

Input Splitting: The input is split into tokens, such as words, subwords, or characters. Token ID Conversion: These tokens are converted into token IDs. Embedding Vectors: Token IDs are converted into embedding vectors. Positional Encoding: Positional encoding values are added to the embedding vectors, resulting in input vectors for the model.

2. Layer Processing:

Layer Passing: The input vectors are passed through the layers of the model sequentially. Shape Consistency: The output from each layer has the same shape as the input, but the values of embedding vectors evolve with

each layer. Batch Processing: During training, a batch of inputs is processed together to optimize resource usage.

3. Prediction and Output:

Final Layer Output: After the final layer, the output consists of a series of embedding vectors. Token Prediction:

```
- Linear Layer: Converts each embedding vector into a new vector with prediction scores for each token
- Softmax Conversion: Converts the prediction scores into probabilities using the softmax function. This
```

By following this structure, we ensure that the Transformer model processes and generates data accurately, leveraging the power of multi-head attention and feedforward networks. This design allows the model to handle complex tasks like language modeling, translation, and more.

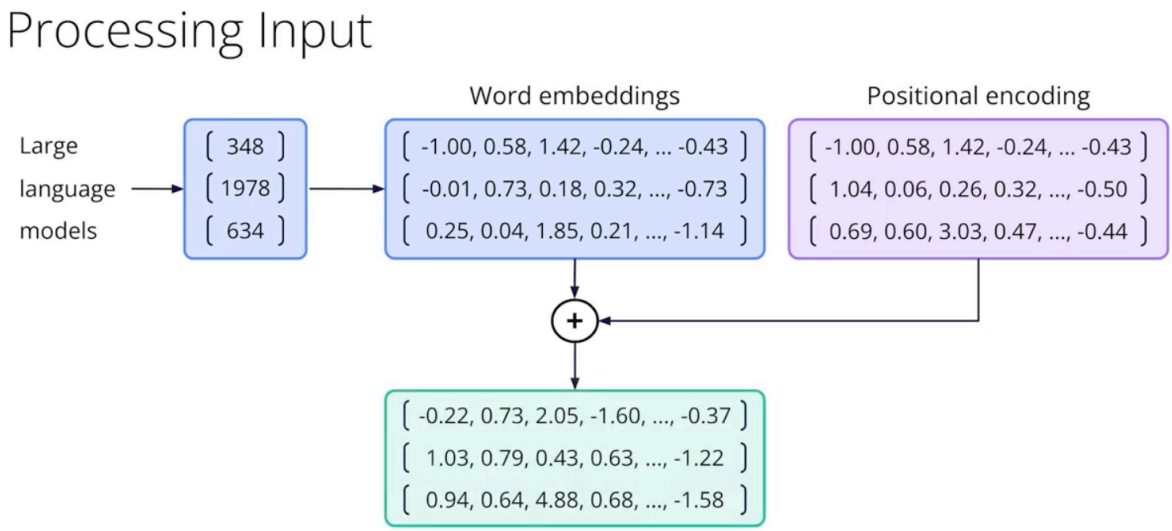

Figure 28: title

Building Blocks of the Model

Token Embedding

- **Purpose**: Converts input token IDs into embedding vectors.
- **Implementation**: Using nn.Embedding, which maps each token ID to its corresponding embedding vector.
- **Parameters**:
  - Number of embedding vectors (vocabulary size).
  - Dimensionality of the embedding vectors.

Positional Encoding

- **Purpose**: Adds positional information to the embedding vectors, helping the model understand the order of tokens.
- **Implementation**: Another nn.Embedding layer, but with the number of vectors equal to the context size (the maximum number of tokens in a sequence).

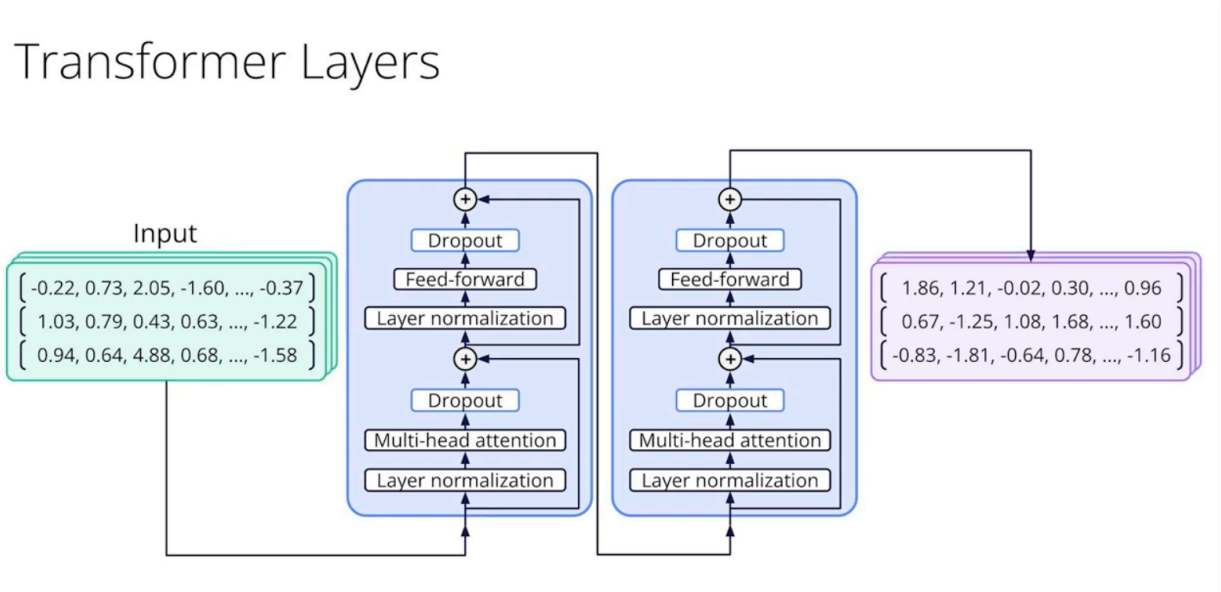

Figure 29: title

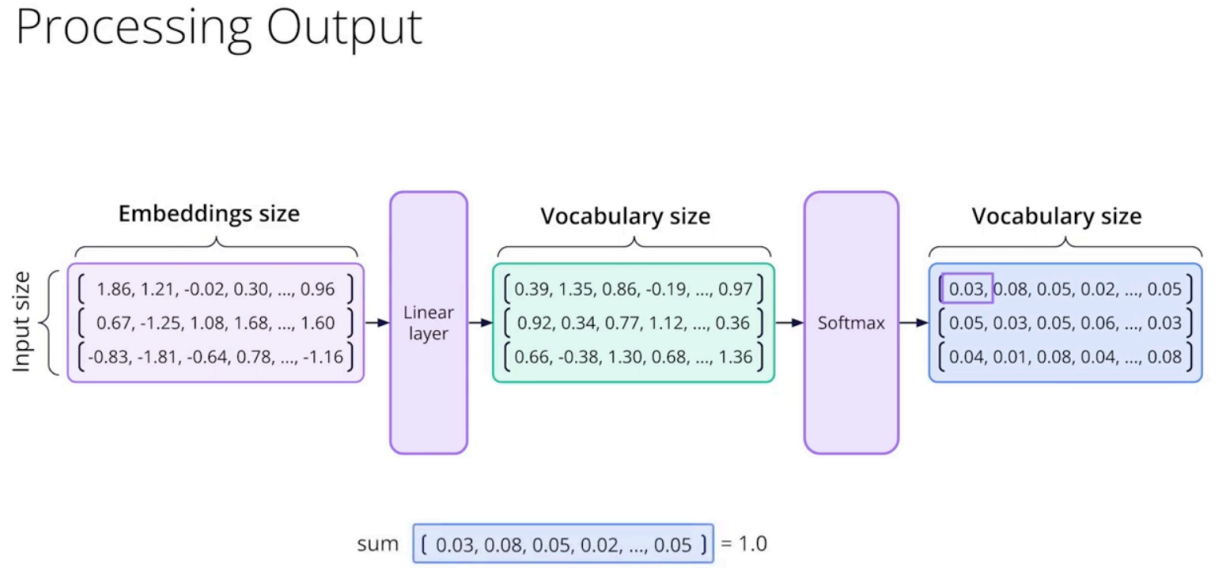

Figure 30: title

Transformer Blocks

- **Creation**: We create multiple Transformer blocks, each containing a multi-head attention layer and a feedforward layer.
- **Combination**: These blocks are combined sequentially, allowing data to pass through each layer.

Final Layer Normalization and Linear Layer

- **Layer Normalization**: Normalizes the output from the last Transformer block.
- **Linear Layer**: Converts the final output vectors into prediction scores for each token in the vocabulary, determining which token should be generated next.

Training Workflow

1. Data Loading

- **Data Loader**: In each iteration, we use a data loader to load a batch of inputs and targets.
  - **Inputs**: Tensor of token IDs representing the input text.
  - **Targets**: Tensor of token IDs representing the expected output.

2. Model Prediction

- **Forward Pass**: The batch of inputs is passed through the model, which generates predictions.
  - **Predictions**: The model outputs logits, which are raw prediction scores for each token in the vocabulary.

3. Loss Calculation

- **Loss Function**: We calculate the loss by comparing the model's predictions with the actual targets.
  - **Purpose**: The loss function quantifies how far the model's predictions are from the actual values.

4. Backpropagation

- **Gradient Calculation**: With the calculated loss, we perform backpropagation to compute the gradients of the loss with respect to the model parameters.

5. Parameter Update

- **Optimizer**: Using the optimizer, we update the model's parameters to minimize the loss.
  - **Iteration Completion**: This completes a single iteration of training.

6. Repeating the Process

- **Epochs**: The training process is repeated for many iterations and epochs, using different batches of data to improve the model's performance.

By following this workflow, the model learns to generate more accurate text outputs over time. Each batch of data helps the model fine-tune its weights and biases, gradually improving its predictions.

Training Setup

Hyperparameters

- **Batch Size**: Set to 64, meaning each training step processes 64 sequences at a time.
- **Iterations**: Training runs for 5,000 iterations.
- **Evaluation Frequency**: Evaluate the model every 100 iterations by generating text.
- **Learning Rate**: Controls how much to adjust model parameters during each training step.

Data Preparation

- **Tokenization**: The text data is tokenized and converted into token IDs. The encode method returns a one-dimensional tensor where every element is a token ID, and we put the tensor on the selected device, which will be a GPU if available.

```
tokenized_text = tokenizer.encode(text).to(device)
```

- **Dataset and Data Loader**:
  - **Dataset**: Created using the tokenized text.
  - **Random Sampler**: Randomly selects training examples, specifying the number of examples based on training iterations and batch size.
  - **Data Loader**: Iterates over selected training examples using the specified batch size and random sampler.

```
dataset = TokenIdsDataset(tokenized_text, block_size=64)
sampler = RandomSampler(dataset, replacement=True, num_samples=train_)
data_loader = DataLoader(dataset, batch_size=batch_size, sampler=samp
```

Optimizer

- **AdamW Optimizer**: Used to update model parameters during training, known for better training results. The learning rate is set based on our chosen hyperparameter.

```
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
```

Training Loop

- **Mini-Batch Iteration**: Iterate over mini-batches using the data loader.
- **Training Mode**: The model is switched to training mode for each iteration.
- **Forward Pass**: The input from each mini-batch is passed through the model, returning logits.
- **Loss Calculation**:
  - **Logits and Targets Reshaping**: Necessary to fit the shape expected by the cross-entropy function.
  - **Cross-Entropy Loss**: Calculates the loss for the step, comparing model predictions to the actual targets.
- **Backpropagation**: Compute gradients based on the loss.
- **Parameter Update**: The optimizer updates model parameters.
- **Gradient Reset**: Before the next step, gradients are reset to avoid accumulation.

Evaluation and Progress Monitoring

- **Text Generation**: Every 100 iterations, the model generates text from a simple prompt (e.g., a newline character) to evaluate training progress.
- **Loss Tracking**: The decreasing loss indicates that the model is learning to generate more accurate text.

Training Progress

- **Initial Outputs**: Early in training, the model generates random text.
- **Gradual Improvement**: As training progresses, the model starts generating text that resembles the training data, including character names and dialogues.
- **Further Training**: Continued training leads to more coherent text generation, eventually resembling Shakespearean style.

By following this training process, the model learns to generate text similar to the input data. The more iterations we train it, the better it becomes at generating coherent and stylistically accurate text.

Calculating Validation Loss

These are the steps for calculating the validation loss:

Split the Dataset: Separate the data into training and validation datasets to evaluate the model's performance during training. Create a Validation Dataset and DataLoader: Prepare the data for evaluation. Calculate Validation Loss: Implement the logic to compute the validation loss, which provides insight into how well the model performs on the validation dataset. Update the Training Loop: Integrate validation loss calculation into the training loop and visualize training and validation loss over time. The starter code provided includes the essential components of a PyTorch model, including data tokenization, model definition, and a basic training loop. Your task is to complete the implementation, focusing on the validation aspect of the model's training process.

Splitting the Dataset

To do this, we first need to split our dataset into training and validation datasets. We tokenize the input text just as we did before and then calculate how many tokens we want to use for training. Given this ratio, we take this number of tokens as training data and use the rest as validation data. Having training and validation data, we then create a validation dataset and a validation data loader.

```
tokenized_text = tokenizer.encode(text).to(device)
train_count = int(train_split * len(tokenized_text))
train_data, validation_data = tokenized_text[:train_count], tokenized_text[train_count:]
train_dataset = TokenIdsDataset(train_data, config["context_size"])
validation_dataset = TokenIdsDataset(validation_data, config["context_size"])
```

Validation Loss Calculation Next, we implement a function to calculate the validation loss. First, we switch our model from training mode into evaluation mode to run our model for inference, which, for example, deactivates the Dropout layers we've added to our model. Keep in mind that we switch our model back to training mode at the beginning of every training iteration.

```
@torch.no_grad()
def calculate_validation_loss(model, batches_num):
    model.eval()
    total_loss = 0
```

We then create an iterator from our validation dataset and run a loop for the specified number of batches. For every batch, we perform similar steps to those in training: we first get the inputs and targets, pass these inputs to the model to get logits, and then reshape logits and targets to calculate the validation loss using the cross-entropy function. After this, we add the loss to the total validation loss for every iteration. When the loop is finished, we compute the average loss by dividing the total loss by the number of batches we processed, and then we return the average loss.

```
    validation_iter = iter(validation_dataloader)

    for _ in range(batches_num):
        input, targets = next(validation_iter)
        logits = model(input)

        logits_view = logits.view(batch_size * config["context_size"], config["vocabulary_size"])
        targets_view = targets.view(batch_size * config["context_size"])
        loss = F.cross_entropy(logits_view, targets_view)
        total_loss += loss.item()

    average_loss = total_loss / batches_num

    return average_loss
```

Plotting Training and Validation Loss

Finally, we update our code to draw a chart of training and validation losses. We append the training loss and training step to two arrays, using the item method to convert a tensor to a number for every training step. We do the same for the validation loss. The reason we need to store the step number with loss values is that we calculate training loss on every step and validation loss only every ten steps, and we need to preserve step numbers for Matplotlib to correctly plot these charts and align values accurately.

```
plot_output = widgets.Output()
display(plot_output)

def update_plot(train_losses, train_steps, validation_losses, validation_steps):
    with plot_output:
        clear_output(wait=True)
        plt.figure(figsize=(7, 5))
        plt.plot(train_steps, train_losses, label='Training Loss')
        plt.plot(validation_steps, validation_losses, label='Validation Loss')
        plt.title('Training and Validation Loss')
        plt.xlabel('epoch')
        plt.legend(loc='center left')
        plt.grid(True)
        plt.show()
```

If we now run our model, and I'll rerun this notebook, as you can see, we've got our training and validation loss chart. If we wait for a few dozen iterations, we will see two lines on the chart, one for the training loss and another for the validation loss.

Training and Validation Loss Calculation Step Summary

1. Splitting the Dataset

To calculate validation loss, split the dataset into training and validation datasets. Tokenize the input text and determine the number of tokens for training based on the desired ratio. The rest of the tokens are used for validation. Create datasets and data loaders for both training and validation data.

2. Validation Loss Calculation Function

Switch the model to evaluation mode to deactivate Dropout layers. Use an iterator to process batches from the validation dataset. For each batch: Extract inputs and targets. Pass inputs through the model to obtain logits. Reshape logits and targets to calculate loss using the cross-entropy function. Aggregate the loss to compute the total validation loss. Average the total loss by the number of processed batches to obtain the validation loss.

3. Plotting Training and Validation Loss

Store training and validation losses along with the corresponding steps. Use Matplotlib to plot the losses, ensuring step numbers align with loss values.

Now let's take a look at how we can calculate validation loss for our model during training. To do this, we first need to split our dataset into training and validation datasets. We tokenize the input text just as we did before and then calculate how many tokens we want to use for training. Given this ratio, we take this number of tokens as training data and use the rest as validation data. Having training and validation data, we then create a validation dataset and a validation data loader.

Next, we implement a function to calculate the validation loss. First, we switch our model from training mode into evaluation mode to run our model for inference, which, for example, deactivates the Dropout layers we've added to our model. Keep in mind that we switch our model back to training mode at the beginning of every training iteration.

We then create an iterator from our validation dataset and run a loop for the specified number of batches. For every batch, we perform similar steps to those in training: we first get the inputs and targets, pass these inputs to the model to get logits, and then reshape logits and targets to calculate the validation loss using the cross-entropy function. After this, we add the loss to the total validation loss for every iteration. When the loop is finished, we compute the average loss by dividing the total loss by the number of batches we processed, and then we return the average loss.

Finally, we update our code to draw a chart of training and validation losses. We append the training loss and training step to two arrays, using the item method to convert a tensor to a number for every training step. We do the same for the validation loss. The reason we need to store the step number with loss values is that we calculate training loss on every step and validation loss only every ten steps, and we need to preserve step numbers for Matplotlib to correctly plot these charts and align values accurately.

If we now run our model, and I'll rerun this notebook, as you can see, we've got our training and validation loss chart. If we wait for a few dozen iterations, we will see two lines on the chart, one for the training loss and another for the validation loss.

Using Pre-trained Transformers

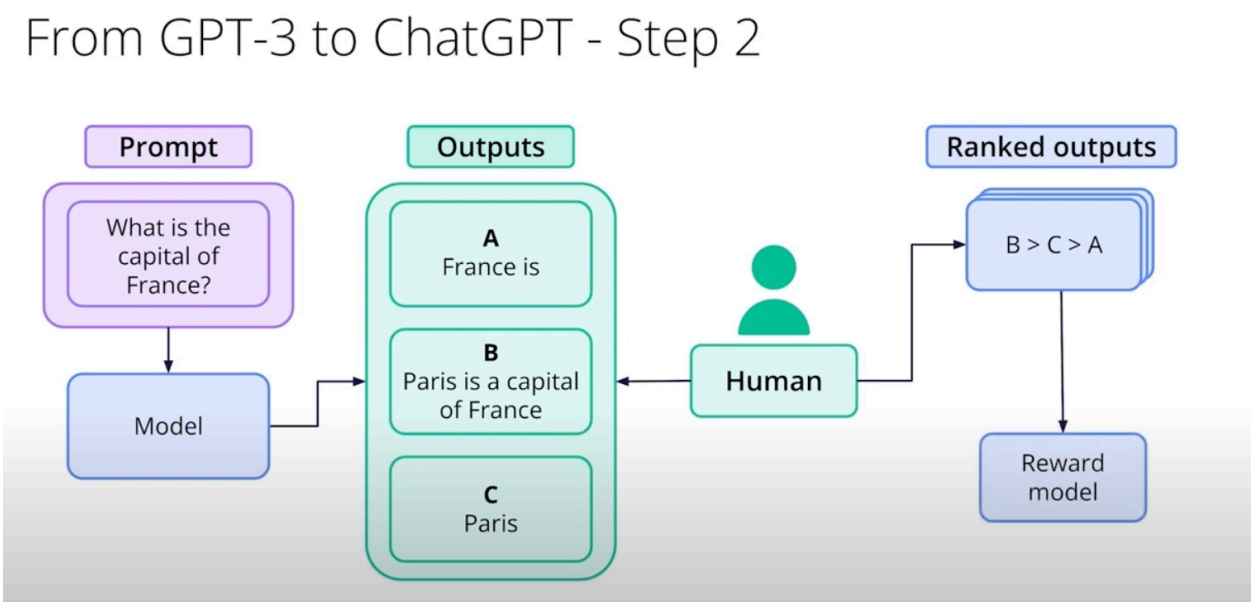

Figure 31: title

Model Scaling and Capabilities One trend you can see is that the bigger the model is, the more capable it usually becomes, and there are already some models with one trillion parameters. However, scaling models is not the only thing needed to achieve capabilities like those of products such as ChatGPT. If we train a model that can simply predict the next word, it will only be able to generate text but won't be able to answer questions like ChatGPT. This limitation does not depend on the amount of training data used or the model size.

Example of Model Limitations To give you an example, I've sent the following prompt to one of the older versions of GPT-3, which was trained to generate text instead of answering questions. As you can see, instead of answering the question, it started generating text related to the question. After being trained on a larger text corpus, it can only generate text that could surround a text that looks like the provided prompt. In contrast, if you send this question to ChatGPT, it would return an answer like 'Paris' and possibly provide some additional information.

Training ChatGPT to Answer Questions So, how did OpenAI train a model that can answer questions? Let's take a high-level overview. They first trained a GPT model similar to what we've built but bigger and using much more data. Then, they used a dataset of questions and answers. With this dataset, they updated the weights of the trained GPT model to create a model that can answer questions. This new model, produced as a result of this process, was called GPT-3.5 and was used by the first version of ChatGPT. This process of taking a model trained for one task and modifying its weights to work for a different task is called fine-tuning, and it is used not just for LLMs but for other deep learning models as well.

Fine-Tuning Process To fine-tune a model to perform question answering, OpenAI went through the following steps:

They collected a dataset with different prompts and questions. They had humans answer these questions and organized the human-provided answers into a dataset. This dataset was then used to fine-tune the GPT-3 model to train it to answer questions. Further Refinement with Reward Models After this, they went through a different process. Using prompts, they sampled several outputs from the model to generate multiple answers for the same question. Then, people ranked these answers against each other. Using these ranking results, OpenAI created another dataset. This dataset of ranked answers was used to train another model called the reward model, which can calculate how good an answer to a prompt is. Finally, using the reward model, they fine-tuned the GPT-3 model to give answers that are more closely aligned with a reward policy.

Decoder-only Architecture A decoder-only transformer is like a predictive text system that looks at what's already been written to figure out what should come next. Imagine writing a text message where your phone suggests the next word - it only uses what you've already typed to make its predictions. This is exactly how decoder-only transformers like GPT work - they only look at the previous context to generate the next piece of text.

Technical Details of Decoder-Only Architecture:

1. **Core Components**

- Uses only the decoder part of the original transformer architecture
- Employs causal (masked) self-attention to prevent looking ahead
- Maintains autoregressive property (each token depends only on previous tokens)

2. **Key Characteristics**

- **Masked Attention**: Only attends to previous tokens
- **Single Stack**: Unlike encoder-decoder models, uses a single transformer stack
- **Unidirectional**: Information flows only from left to right
- **Parameter Efficiency**: Fewer parameters than full encoder-decoder models

3. **Popular Examples**

- GPT (all versions)
- BLOOM
- LLaMA
- Claude (base architecture)

4. **Primary Applications**

- Text generation
- Language modeling
- Completion tasks
- Conversational AI

5. **Advantages**

- Simpler architecture
- Efficient training

- Good at generative tasks
- Strong zero-shot capabilities

6. **Limitations**

- No bidirectional context
- Less suited for tasks requiring full context (like translation)
- May struggle with long-range dependencies

The decoder-only architecture has become particularly popular for large language models due to its simplicity and effectiveness in generative tasks.

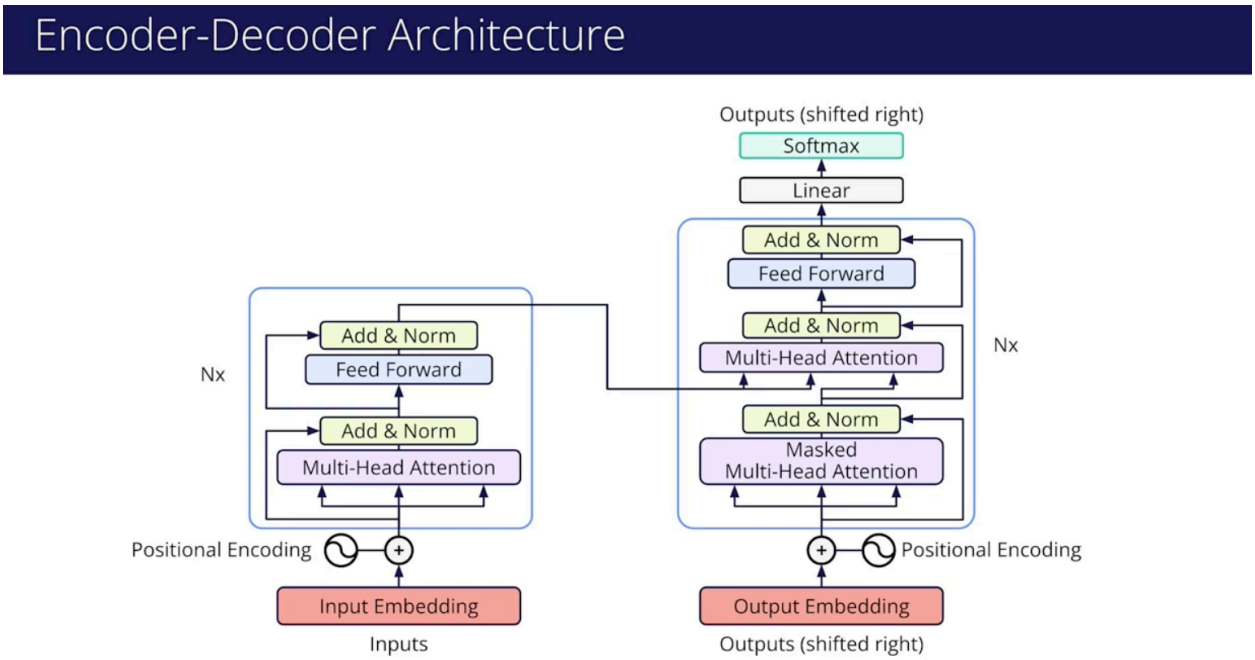

Figure 32: title

Origin of Transformers Transformers were initially introduced in a paper called "Attention Is All You Need." This paper described an architecture more complex than what we implemented. This architecture was created for a different purpose: instead of generating text, the authors focused on translating text from one language to another.

Key Components of Transformer Architecture Decoder Component: The right part of the architecture, similar to our model, includes: Embedding layer Positional encoding Masked multi-head attention (using a triangular mask to modify attention scores) Feedforward layer with residual connections Nx Repeated Blocks: This block is repeated multiple times, similar to our model. Encoder Component: The left part, absent in our model, was used to encode information about an input text for translation. The encoder uses multi-head attention (without masking) and processes the entire input. Interaction Between Encoder and Decoder: The encoder processes the input text, and its final layer's output is passed to the decoder layers. The decoder generates the next token based on the encoded input and its partially generated output. Example: Translation from English to French Tokenization: The English text is tokenized and converted into token IDs. Encoding: Token IDs are passed to the encoder to build a representation of the phrase. Decoding: The process starts with a "start of the sequence" token sent to the decoder. The decoder reads its input and the encoder's output to produce the next token ID of the translated phrase. This process repeats, generating more tokens until an "end of sequence" token is produced. The final token IDs are converted into a human-readable sentence. Different Network Architectures Using Encoders and Decoders Decoder-only models: Only include the decoder part, without an encoder. Suitable for text generation tasks. Examples:

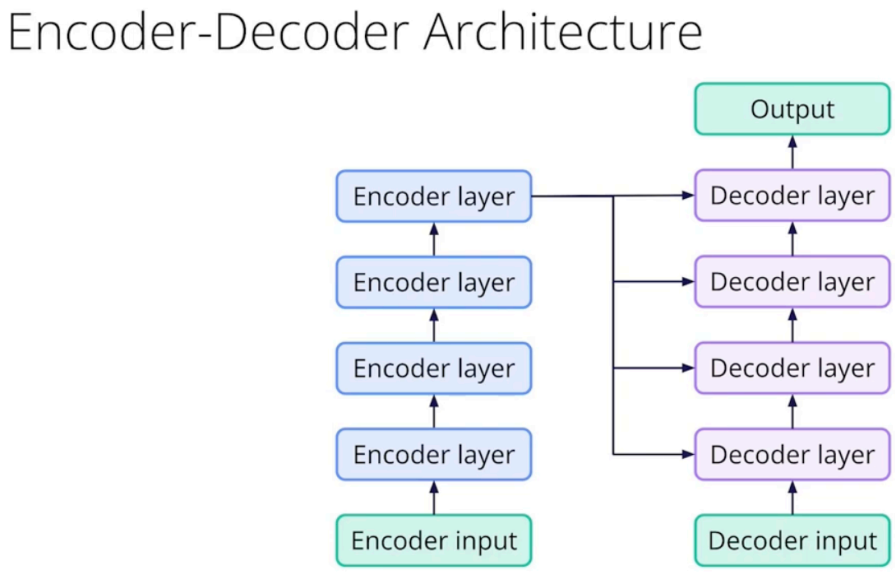

Figure 33: title

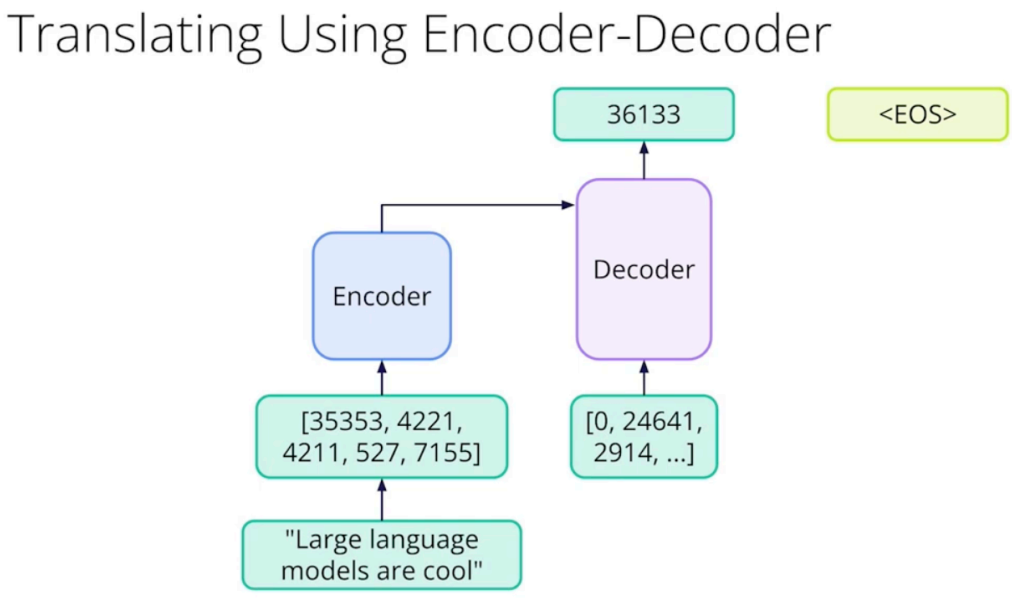

Figure 34: title

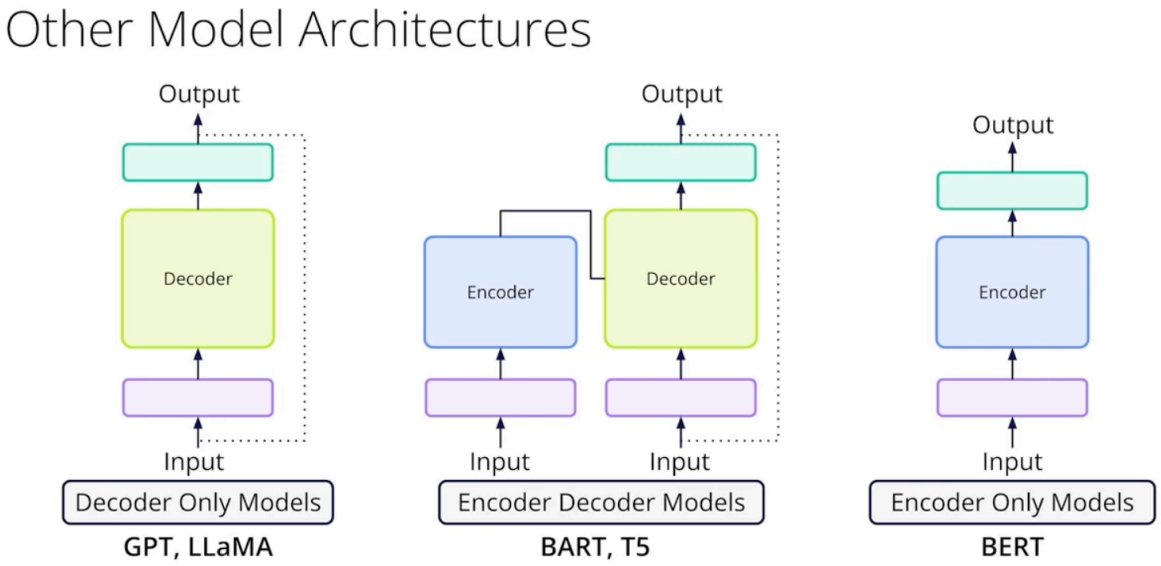

Figure 35: title

GPT and LLaMA. Encoder-decoder models: Include both encoder and decoder blocks. Ideal for tasks like translating text into another language. Examples: BERT and T5.

Encoder-only models: Lack a decoder, using only the encoder to process inputs. Useful for text classification tasks like sentiment analysis or spam detection. Example: BERT.

Using Pre-Trained Transformer Models While it is important to understand how large language models work under the hood, it is unlikely you will need to create one from scratch for your applications. Training an LLM from scratch is challenging for several reasons:

Data and Resources: Training these models requires huge amounts of training data and a lot of computational resources. Cost: These models are very expensive to train. For example, the cost of training the GPT-4 model was more than $100 million. Expertise: Developing new models requires a specialized skill set, especially if you aim to implement a state-of-the-art model. Hugging Face: A Resource for Pre-Trained Models Fortunately, instead of creating a model from scratch, we can use existing models created by companies and researchers from all around the world. To do this, we will use the Hugging Face library, which is one of the most popular NLP libraries for Python. At the moment of this recording, it has a whopping 27,000 stars on GitHub. It also has a huge community contributing to the project, developing new large language models, and improving its features.

Hugging Face simplifies building applications using LLMs but is not limited to this. It also provides tools for developing new LLMs. Hugging Face is a collection of multiple components that work together. It provides an API that allows us to build applications using LLMs. We can access a large collection of pre-trained models that we can use in our applications. It also provides access to datasets for different purposes that we can use to train our models or improve existing models. Finally, it provides the Spaces project, which allows us to quickly create machine learning demos and applications.

API and Capabilities Hugging Face provides a set of flexible and powerful APIs that we can use to build our AI applications. Using their API, we can do things like access existing models for a variety of tasks such as text generation, text summarization, and so on. We can train new models or fine-tune existing models from their library of models. Fine-tuning is a process that allows us to take an existing model and update its weights to make it better for a specific task for which the model wasn't originally trained. We

can also process text for LLMs, for example, by tokenizing text using an existing tokenizer or creating our own tokenizer. Finally, we can deploy our LLMs.

Beyond Text: Images and Audio Hugging Face is not limited to working with just text; it also includes APIs to work with images and audio.

Comparing Hugging Face and OpenAI You might be wondering how Hugging Face is different from OpenAI. Both provide models and allow us to build applications using them, but they are very different companies. Hugging Face is an open-source product; we can run models from Hugging Face locally on our laptops or on servers that we control. Hugging Face models are also free to use. On the other hand, OpenAI, despite the name, provides closed-source models. They only allow us to access their models via their API, and these models run on servers that OpenAI operates. Additionally, we need to pay to access advanced OpenAI models.

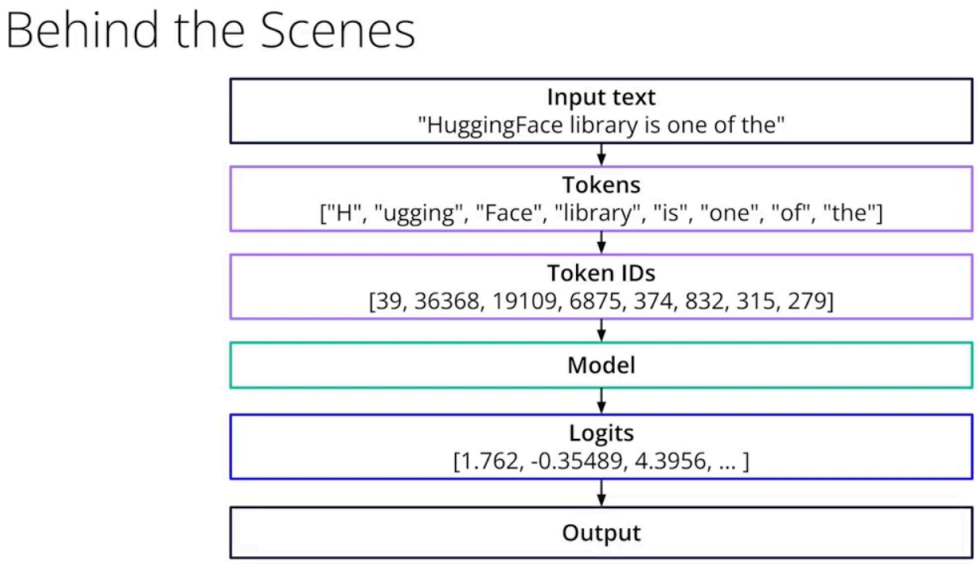

Figure 36: title

Hugging Face Library When it comes to using Hugging Face models, we have two options:

High-Level API: This is accessible using the pipeline function and is user-friendly. Low-Level API: This offers more flexibility and control over the model's usage. Since this is a beginner-friendly lesson, we will focus just on using the high-level API, but keep in mind that Hugging Face API has much more to offer.

Text Generation with Hugging Face To use a large language model with Hugging Face:

Import the Pipeline Function: Import the pipeline function from the Transformers package. Specify the Task: Call the pipeline function and provide the name of the task, such as "text-generation." This function returns another function that we can use to perform the selected task. Provide a Prompt: For text generation, specify a prompt that will be used to generate the text. To generate text using this prompt:

Call the generator function returned by the pipeline call. Provide parameters such as the prompt, the maximum length of the output, and the number of sequences to generate. Hugging Face's Versatility Hugging Face supports a wide range of NLP-related tasks, including:

Text Generation: As described above. Question Answering: Finding answers to questions based on a given context. Text Summarization: Summarizing large pieces of text. Translation: Translating text between

different languages. Text Classification: Classifying text into categories. In addition to NLP tasks, Hugging Face also supports:

Audio Tasks: Such as audio classification and text-to-speech conversion. Image and Video Tasks: Including image classification, video classification, and image segmentation. Example: Image Classification with Hugging Face Here's how to perform image classification using Hugging Face:

Specify the Task: Use the pipeline function, specifying the task as "image-classification." Pass an Image: Provide an image file or an image URL for classification. Interpret the Output: The output is a list with each element containing a label and an associated probability, indicating what the model thinks the image represents. For example, it might classify an image as likely being a cat.

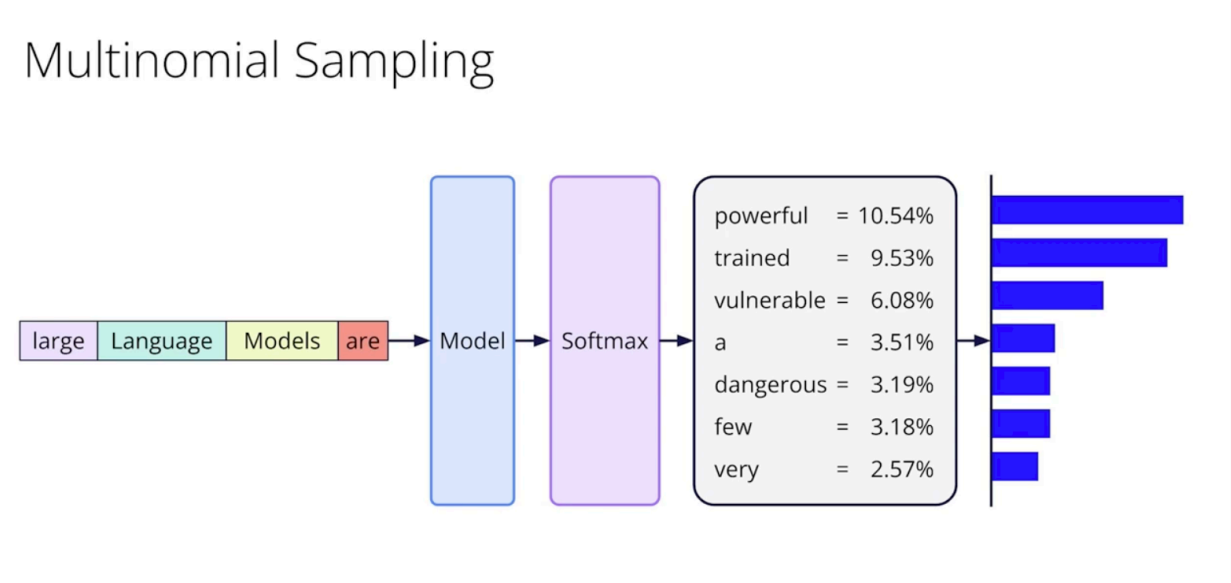

Figure 37: title

Text Generation Options To avoid some possible confusion, there are two types of parameters:

model parameters, which are set during model training, and generation parameters, which can affect the result of text generation when we use a model after it is trained. Maximum Number of Tokens We have already seen one parameter, which is the maximum number of tokens the text generation process should return. In Hugging Face, this is called the max_length parameter. As you can imagine, the higher this parameter is, the longer the output the text generation process will produce. However, one gotcha is that it defines a maximum number of tokens, and text generation can return an output that is shorter than this value. During the text generation process, a model can emit a special "end of sequence" token that will stop the generation process even before it reaches the specified limit.

Greedy Decoding One simple option for generating text is to select the token with the highest probability every time a new token is generated. This approach, called "greedy decoding," picks the word with the highest probability and adds it to the input, repeating the process. However, this can lead to boring and repetitive results.

Multinomial Sampling To avoid repetitive results, we can introduce randomness by picking less probable tokens. This method, known as " multinomial sampling," uses the output of the softmax layer as a probability distribution to randomly select tokens. The higher the probability of a token, the more likely it will be selected. This approach provides a more varied and interesting output.

Beam Search Beam search is an algorithm used to generate multiple sequences at the same time, evaluating them to find the best result. Instead of generating a single sequence, beam search keeps "n" best sequences

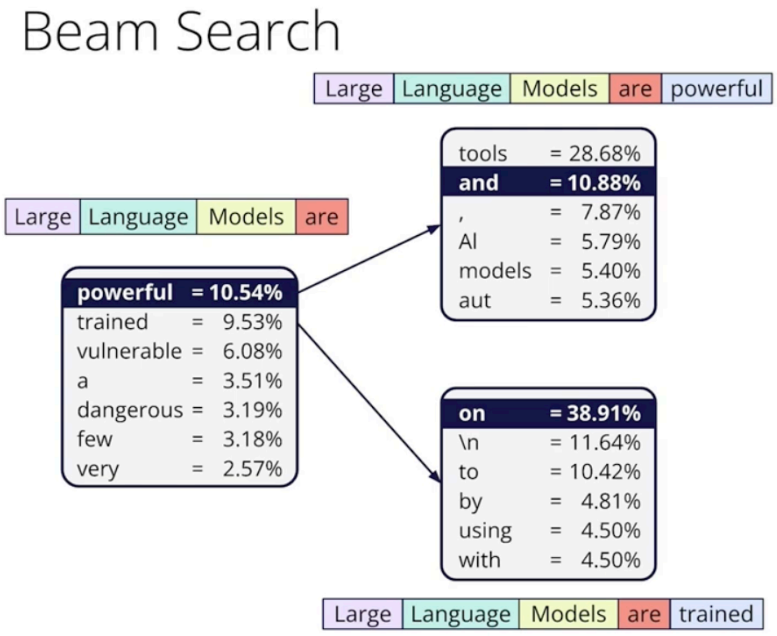

Figure 38: title

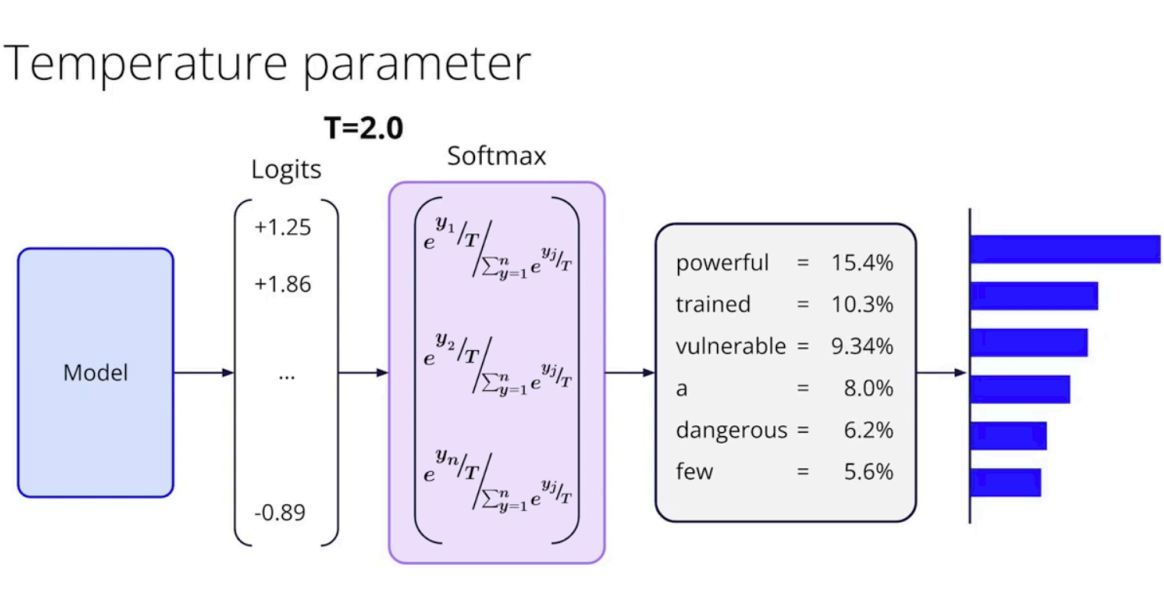

Figure 39: title

at each step, generating new sequences from these and repeating the process. This method helps in finding more optimal sequences by considering multiple possible outputs simultaneously.

Top-K Sampling The top_k parameter specifies the number of tokens with the highest probability to consider when selecting the next token. For example, setting top_k to 3 means selecting the next token from the top three most probable tokens. Higher top_k values can lead to more creative outputs but may also produce less coherent text.

Top-P (Nucleus) Sampling The top_p parameter specifies the cumulative probability of the most probable tokens to consider when selecting the next token. This method ensures that only tokens that collectively meet a certain probability threshold are considered, allowing for dynamic adjustment of the number of tokens based on their probabilities.

Temperature The temperature parameter adjusts the probability distribution returned by the softmax layer. Lowering the temperature increases the chances of selecting tokens with higher logits, making the output more deterministic. Increasing the temperature flattens the probability distribution, allowing for more diverse token selection.

Using Parameters with Hugging Face In Hugging Face, these parameters can be set when using the generator function. For example, do_sample enables multinomial sampling, and num_beams specifies the number of sequences in beam search. Parameters like top_k, top_p, and temperature control the randomness and creativity of the generated output, allowing for fine-tuning the behavior of the language model according to specific requirements.