Unit 09: Introduction to Neural Networks

CCNY EAS 42000/A4200, Fall 2025

Prof. Spencer A. Hill

Dept. of Earth & Atmospheric Sciences
City College of New York

December 1,3,8 2025

Unit 09: Introduction to Neural Networks

Today's goals

1. describe what a *feedforward* **neural network** is and how it is *trained*

Today's goals

1. describe what a *feedforward* **neural network** is and how it is *trained*
2. see how neural networks extend ideas you know from OLS linear regression into more flexible and nonlinear models

Today's goals

1. describe what a *feedforward* **neural network** is and how it is *trained*
2. see how neural networks extend ideas you know from OLS linear regression into more flexible and nonlinear models
3. understand *overfitting*, why it's important, and some techniques for avoiding it

Motivation

Neural networks are a core tool for prediction and pattern recognition across many fields, including EAS.

Motivation

Neural networks are a core tool for prediction and pattern recognition across many fields, including EAS.

Examples: forecasting weather, emulating expensive model components, predicting ENSO, many many others.

Motivation

Neural networks are a core tool for prediction and pattern recognition across many fields, including EAS.

Examples: forecasting weather, emulating expensive model components, predicting ENSO, many many others.

But now: step back to focus on the conceptual machinery: what neural networks are, how they learn from data, and why overfitting is a central concern.

Unit 09: Introduction to Neural Networks

You've actually already seen a model that "learns" a relationship between predictors and responses. What is it?

You've actually already seen a model that "learns" a relationship between predictors and responses. What is it?

Linear regression! Recall:

$$\tilde{y} = \alpha + \beta x,$$

where α and β are determined ("learned") via ordinary least squares to *minimize the mean-square error*.

While linear regression emphasizes simplicity and clarity, ML emphasizes *predictive performance*

But still, the key ideas are the same:

We choose a function, we define a numerical measure of prediction error, and we adjust the model's parameters to reduce that error on observed data.

Schematic of a neural network

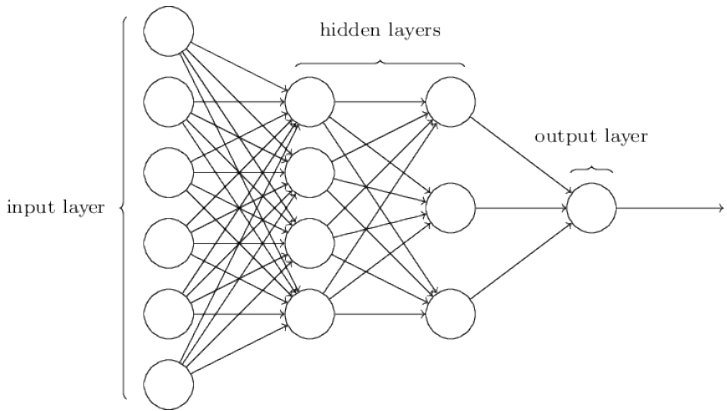

Figure: http://neuralnetworksanddeeplearning.com/images/tikz11.png

The *particular* case shown: 6 inputs, 2 hidden layers (with 4 and 3 neurons, respectively), and one single output

Schematic of *linear regression* in the same style as used for neural networks

[on the board: 1 input (the x value) and 1 output (the corresponding y value)]

We will focus on ***supervised learning***, meaning that *we know the correct answer* y_i for each input x_i

In other words, we have on hand pairs of inputs x and their corresponding outputs y:

$$(x_i, y_i), \quad i = 1, \ldots, n.$$

We will focus on ***supervised learning***, meaning that *we know the correct answer* y_i for each input x_i

In other words, we have on hand pairs of inputs x and their corresponding outputs y:

$$(x_i, y_i), \quad i = 1, \ldots, n.$$

The y_i values are often called the ***labels***, and the overall data x, y is called *labeled* data.

We will focus on ***supervised learning***, meaning that *we know the correct answer* y_i for each input x_i

In other words, we have on hand pairs of inputs x and their corresponding outputs y:

$$(x_i, y_i), \quad i = 1, \ldots, n.$$

The y_i values are often called the ***labels***, and the overall data x, y is called *labeled* data.

(As contrasted with *unsupervised learning*, which we will not cover)

We then construct some model, f_θ, that computes its own output, $\tilde{y}_i$, for each input x_i

That is:

$$\tilde{y}_i = f_\theta(x_i),$$

where θ is the set of all of the adjustable parameters of the model.

We then construct some model, f_θ, that computes its own output, $\tilde{y}_i$, for each input x_i

That is:

$$\tilde{y}_i = f_\theta(x_i),$$

where θ is the set of all of the adjustable parameters of the model.

In simple OLS linear regression, $f_\theta(x) = (\alpha + \beta x)$, with $\theta = \{\alpha, \beta\}$.

We then construct some model, f_θ, that computes its own output, $\tilde{y}_i$, for each input x_i

That is:

$$\tilde{y}_i = f_\theta(x_i),$$

where θ is the set of all of the adjustable parameters of the model.

In simple OLS linear regression, $f_\theta(x) = (\alpha + \beta x)$, with $\theta = \{\alpha, \beta\}$.

The key distinction between non-ML and ML models is in *how the parameter values* θ *get determined*.

Non-ML models: parameter values computed *from pre-existing formulas*.
ML models: parameter values *determined* directly from the data.

For a non-ML example, in simple OLS the parameter values are *always*: $\alpha = \hat{\mu}_y - \beta\hat{\mu}_x$
$\beta = \hat{\text{cov}}(x, y)/\hat{\sigma}_x^2$.

We can only train the model using data on hand, but we want it to work well for *unseen* data too

In principle, we would like to choose parameters that minimize the expected loss on new, unseen data.

We can only train the model using data on hand, but we want it to work well for *unseen* data too

In principle, we would like to choose parameters that minimize the expected loss on new, unseen data.

In practice, we have only a finite sample, so we minimize the average loss over the training set, denoted J:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L\big(f_\theta(x_i), y_i\big),$$

where L is the ***loss function***.

We can only train the model using data on hand, but we want it to work well for *unseen* data too

In principle, we would like to choose parameters that minimize the expected loss on new, unseen data.

In practice, we have only a finite sample, so we minimize the average loss over the training set, denoted J:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L\big(f_\theta(x_i), y_i\big),$$

where L is the ***loss function***.

This is no different than e.g. linear regression: we "train" a simple OLS model on the data we have, and *make assumptions* about how well it generalizes beyond those points.

Training an ML model consists of tuning the values of its parameters (θ) to make the predictions $\hat{y}_i$ match the observed targets y_i as well as possible

But what does “as well as possible” mean?

Training an ML model consists of tuning the values of its parameters (θ) to make the predictions $\hat{y}_i$ match the observed targets y_i as well as possible

But what does “as well as possible” mean?

Answering this requires specifying a ***loss function*** (a.k.a. *cost function*), L, that takes the correct outputs y_i, the model’s predicted outputs $\tilde{y}_i$, and *quantifies* in some way the extent to which they differ.

Training an ML model consists of tuning the values of its parameters (θ) to make the predictions $\hat{y}_i$ match the observed targets y_i as well as possible

But what does “as well as possible” mean?

Answering this requires specifying a ***loss function*** (a.k.a. *cost function*), L, that takes the correct outputs y_i, the model’s predicted outputs $\tilde{y}_i$, and *quantifies* in some way the extent to which they differ.

Again, in OLS there is a direct analog: the loss function is the *mean square error* MSE: $L(y, \tilde{y}) = (1/N) \sum_{i=1}^{N} (y - \tilde{y})^2$

In fact, the MSE (a.k.a. *quadratic error*) is commonly used in ML models as well!

Unit 09: Introduction to Neural Networks

What Is a Neural Network?

A feedforward neural network is a flexible nonlinear function built by composing simple layers.

What Is a Neural Network?

A feedforward neural network is a flexible nonlinear function built by composing simple layers.

Each layer applies a linear transformation followed by a nonlinear activation function, and the output of one layer becomes the input to the next.

What Is a Neural Network?

A feedforward neural network is a flexible nonlinear function built by composing simple layers.

Each layer applies a linear transformation followed by a nonlinear activation function, and the output of one layer becomes the input to the next.

By stacking layers, neural networks can represent complicated input-output relationships that are difficult or impossible to capture with a single linear model.

Schematic of a neural network

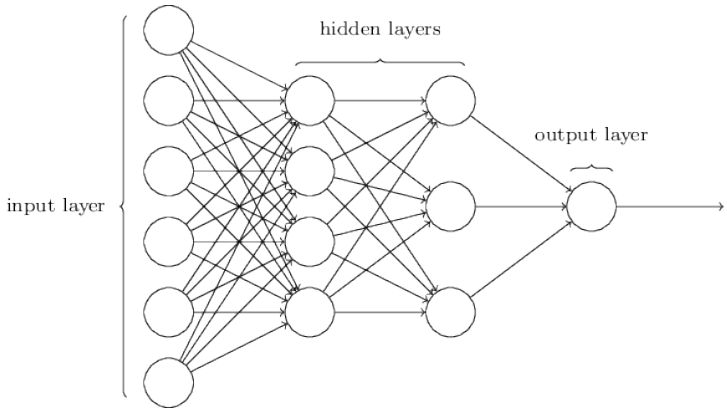

Figure: http://neuralnetworksanddeeplearning.com/images/tikz11.png

Feedforward Architecture

In a fully connected feedforward network, information flows in one direction: from the input layer, through one or more hidden layers, to the output layer.

Feedforward Architecture

In a fully connected feedforward network, information flows in one direction: from the input layer, through one or more hidden layers, to the output layer.

Every unit (or node) in one layer is connected to every unit in the next layer via a weight, and each unit also has an associated bias.

Feedforward Architecture

In a fully connected feedforward network, information flows in one direction: from the input layer, through one or more hidden layers, to the output layer.

Every unit (or node) in one layer is connected to every unit in the next layer via a weight, and each unit also has an associated bias.

Nonlinear activation functions, such as the sigmoid or ReLU, are applied at each hidden unit so that the overall mapping from inputs to outputs is nonlinear.

A Simple Two-Layer Network

Consider a network with one hidden layer. Let x be the input vector, h the hidden activations, and $\hat{y}$ the output.

A Simple Two-Layer Network

Consider a network with one hidden layer. Let x be the input vector, h the hidden activations, and $\hat{y}$ the output.

We can write the forward mapping as

$$h = \sigma(W_1 x + b_1),$$

$$\hat{y} = W_2 h + b_2,$$

where W_1, W_2 are weight matrices, b_1, b_2 are bias vectors, and $\sigma(\cdot)$ is an elementwise activation function.

Why This Increases Modeling Power

A linear regression model cannot capture complex nonlinear patterns.

Why This Increases Modeling Power

A linear regression model cannot capture complex nonlinear patterns.

By composing multiple linear transformations and nonlinear activations, networks can represent highly “curved” *decision boundaries* and intricate functional relationships.

Why This Increases Modeling Power

A linear regression model cannot capture complex nonlinear patterns.

By composing multiple linear transformations and nonlinear activations, networks can represent highly “curved” *decision boundaries* and intricate functional relationships.

The hidden layers can be interpreted as learning intermediate “features” from the raw inputs, instead of relying on features we design manually.

Digit Classification Example

A classic introductory example is handwritten digit recognition using the MNIST dataset.

Digit Classification Example

A classic introductory example is handwritten digit recognition using the MNIST dataset.

Each image is 28×28 pixels, which we can flatten into a 784-dimensional input vector. The network outputs ten numbers representing confidence for digits 0 through 9.

Digit Classification Example

A classic introductory example is handwritten digit recognition using the MNIST dataset.

Each image is 28×28 pixels, which we can flatten into a 784-dimensional input vector. The network outputs ten numbers representing confidence for digits 0 through 9.

We will use Michael Nielsen's implementation of a simple network for this task in the lab, because it provides a clean, fully open example that illustrates all the concepts from today.

To train a neural network, need to quantify how wrong a prediction is, via a ***loss function***

For *regression problems*, a common choice is the *mean-squared error* between predictions and targets.

To train a neural network, need to quantify how wrong a prediction is, via a ***loss function***

For *regression problems*, a common choice is the *mean-squared error* between predictions and targets.

For *classification problems*, such as digit recognition, more common to use a loss based on probabilities, such as the *cross-entropy loss*.

Collecting everything, training a neural network means solving the optimization problem:

$$\theta^* = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} L\big(f_\theta(x_i), y_i\big).$$

Collecting everything, training a neural network means solving the optimization problem:

$$\theta^* = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} L\big(f_\theta(x_i), y_i\big).$$

Here θ includes all weights and biases in all layers. The loss function is differentiable almost everywhere with respect to these parameters.

Gradient descent is an iterative method for minimizing the loss function using its gradient.

Starting from an initial guess $\theta^{(0)}$, we repeatedly update the parameters according to

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)}),$$

where $\eta > 0$ is the learning rate, $J(\theta)$ is the average loss, and ∇_θ is the *gradient* of the loss J with respect to all of the weights and biases.

Gradient descent is an iterative method for minimizing the loss function using its gradient.

Starting from an initial guess $\theta^{(0)}$, we repeatedly update the parameters according to

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)}),$$

where $\eta > 0$ is the learning rate, $J(\theta)$ is the average loss, and ∇_θ is the *gradient* of the loss J with respect to all of the weights and biases.

This moves the parameters in the direction of *steepest local decrease of the loss*, under a local linear approximation.

In **stochastic gradient descent (SGD)**, for computational efficiency we subset the training data into *mini-batches*.

Computing the gradient of the loss over the entire dataset at every step can be computationally expensive for large n.

In **stochastic gradient descent (SGD)**, for computational efficiency we subset the training data into *mini-batches*.

Computing the gradient of the loss over the entire dataset at every step can be computationally expensive for large n.

Stochastic gradient descent (SGD) instead uses the gradient computed on a randomly chosen subset ("minibatch") of the training data at each step.

In **stochastic gradient descent (SGD)**, for computational efficiency we subset the training data into *mini-batches*.

Computing the gradient of the loss over the entire dataset at every step can be computationally expensive for large n.

Stochastic gradient descent (SGD) instead uses the gradient computed on a randomly chosen subset ("minibatch") of the training data at each step.

This reduces computation per update, which can make a huge difference

In **stochastic gradient descent (SGD)**, for computational efficiency we subset the training data into *mini-batches*.

Computing the gradient of the loss over the entire dataset at every step can be computationally expensive for large n.

Stochastic gradient descent (SGD) instead uses the gradient computed on a randomly chosen subset ("minibatch") of the training data at each step.

This reduces computation per update, which can make a huge difference

It also introduces a useful amount of randomness that can help escape shallow local minima and improve generalization.

Why SGD Works in Practice

The “loss surface” can be very bumpy: there is no guarantee that gradient-based methods find a global minimum.

(on the board schematic: 1D loss curve with multiple local minima)

Why SGD Works in Practice

The “loss surface” can be very bumpy: there is no guarantee that gradient-based methods find a global minimum.
(on the board schematic: 1D loss curve with multiple local minima)

Nonetheless, empirical work has shown that many local minima have similar, acceptable loss, and that SGD often finds parameter settings with good predictive performance.

Why SGD Works in Practice

The “loss surface” can be very bumpy: there is no guarantee that gradient-based methods find a global minimum.
(on the board schematic: 1D loss curve with multiple local minima)

Nonetheless, empirical work has shown that many local minima have similar, acceptable loss, and that SGD often finds parameter settings with good predictive performance.

Theoretical understanding is still evolving, but in many applications SGD, combined with good initialization and reasonable hyperparameters, is remarkably effective.

But how do we find the gradient of the loss function, which we need for SGD? **Backpropagation**

To use gradient descent or SGD, we need gradients of the loss with respect to all parameters in the network.

But how do we find the gradient of the loss function, which we need for SGD? **Backpropagation**

To use gradient descent or SGD, we need gradients of the loss with respect to all parameters in the network.

Backpropagation is an efficient algorithm that applies the chain rule of calculus through the computational graph of the network.

But how do we find the gradient of the loss function, which we need for SGD? **Backpropagation**

To use gradient descent or SGD, we need gradients of the loss with respect to all parameters in the network.

Backpropagation is an efficient algorithm that applies the chain rule of calculus through the computational graph of the network.

The main idea is to propagate derivatives backward from the output layer to earlier layers, reusing intermediate quantities computed in the forward pass.

Backpropagation has two steps: a *forward pass* and a *backward pass*

Forward pass: start from the *inputs*, compute the activations of each layer in sequence, and finally compute the loss.

Backpropagation has two steps: a *forward pass* and a *backward pass*

Forward pass: start from the *inputs*, compute the activations of each layer in sequence, and finally compute the loss.

Backward pass:

- Start at the *output layer*, computingf the derivative of the loss with respect to the outputs.
- Then work backward, layer by layer, computing derivatives with respect to activations and then parameters.

Backpropagation has two steps: a *forward pass* and a *backward pass*

Forward pass: start from the *inputs*, compute the activations of each layer in sequence, and finally compute the loss.

Backward pass:

- Start at the *output layer*, computingf the derivative of the loss with respect to the outputs.
- Then work backward, layer by layer, computing derivatives with respect to activations and then parameters.

Result: computed derivatives of the cost function with respect to *every* weight w and bias b, i.e. $\partial L/\partial w$ and $\partial L/\partial b$.

(Technical aside: backpropagation is also *computationally efficient*)

Backpropagation only has to compute each derivative once during the backward pass.

As such, the computational cost of backpropagation is similar to that of evaluating the network once.

(This really matters! If it was extremely computationally expensive, then we wouldn't be able to train nearly as powerful of models.)

Backpropagation is just calculus (the equations of which we won't actually go into)

Backpropagation is not a heuristic or a biologically inspired trick; it is an organized application of the chain rule to a composite function.

Backpropagation is just calculus (the equations of which we won't actually go into)

Backpropagation is not a heuristic or a biologically inspired trick; it is an organized application of the chain rule to a composite function.

Modern software libraries automate this process (often called automatic differentiation), so we rarely need to hand-derive the full set of equations.

Backpropagation is just calculus (the equations of which we won't actually go into)

Backpropagation is not a heuristic or a biologically inspired trick; it is an organized application of the chain rule to a composite function.

Modern software libraries automate this process (often called automatic differentiation), so we rarely need to hand-derive the full set of equations.

However, it's good to keep in mind that training neural networks boils down to computing gradients and taking little steps down a loss curve.

Unit 09: Introduction to Neural Networks

Overfitting: where the model fits the training data very closely but performs poorly on new data

Neural networks have high capacity and can, in principle, memorize the training data if given enough parameters and training cycles.

Overfitting: where the model fits the training data very closely but performs poorly on new data

Neural networks have high capacity and can, in principle, memorize the training data if given enough parameters and training cycles.

It will then fail to *generalize*: it learned all the teeny idiosyncracies of the training set that *won't* carry over to other data

Rather than the more general relationships that actually *are* more general

Overfitting: where the model fits the training data very closely but performs poorly on new data

Neural networks have high capacity and can, in principle, memorize the training data if given enough parameters and training cycles.

It will then fail to *generalize*: it learned all the teeny idiosyncracies of the training set that *won't* carry over to other data

Rather than the more general relationships that actually *are* more general

How to ID Overfitting: low loss on the data it was trained on but high loss on an independent *validation* or *test* dataset.

Why Overfitting Happens

Overfitting arises when the model is “expressive enough” to fit not only the underlying signal but also the noise in the training data.

Why Overfitting Happens

Overfitting arises when the model is “expressive enough” to fit not only the underlying signal but also the noise in the training data.

This is especially problematic when the dataset is small relative to the number of parameters, and/or when the signal-to-noise ratio is low.

Train/validate/test splitting: set aside *validation* and *test* sets of data that are *not* used during training.

During training, track the loss on both the training data *and* validation data as functions of the number of training epochs.

Train/validate/test splitting: set aside *validation* and *test* sets of data that are *not* used during training.

During training, track the loss on both the training data *and* validation data as functions of the number of training epochs.

If validation loss begins to increase while training loss continues to decrease: overfitting!

Train/validate/test splitting: set aside *validation* and *test* sets of data that are *not* used during training.

During training, track the loss on both the training data *and* validation data as functions of the number of training epochs.

If validation loss begins to increase while training loss continues to decrease: overfitting!

So in training, target optimizing the *validation* dataset. Stop once the *validation* loss stops decreasing.
Even if the *training* loss is still improving!

Use the validation data to tune your *hyperparameters*: everything other than the weights and biases

What learning rate gives the best performance? Which batch size? Which number of neurons, or activation function, or loss function?

Use the validation data to tune your *hyperparameters*: everything other than the weights and biases

What learning rate gives the best performance? Which batch size? Which number of neurons, or activation function, or loss function?

These are all *hyperparameters*: things about the model that *you* decide but that are *NOT* changed by the training

Use the validation data to tune your *hyperparameters*: everything other than the weights and biases

What learning rate gives the best performance? Which batch size? Which number of neurons, or activation function, or loss function?

These are all *hyperparameters*: things about the model that *you* decide but that are *NOT* changed by the training

Of course, you try to find the combination of these *hyperparameters* that gives the best performance. Use the validation data for that!

Use the validation data to tune your *hyperparameters*: everything other than the weights and biases

What learning rate gives the best performance? Which batch size? Which number of neurons, or activation function, or loss function?

These are all *hyperparameters*: things about the model that *you* decide but that are *NOT* changed by the training

Of course, you try to find the combination of these *hyperparameters* that gives the best performance. Use the validation data for that!

Then once you've "frozen" your model (i.e. locked in all these hyperparameter values), you test its performance on another dataset that was also withheld from training, called the *test* set.

Strategies we'll discuss for avoiding overfitting

1. ***Regularization***
2. ***Dropout***
3. ***Data Augmentation***

Regularization: penalize very large weight values, because those tend to indicate overfitting

Large-magnitude weights often signal that the network is memorizing idiosyncrasies in the training data

Roughly speaking, uses big weights to find every single wiggle—i.e. fit the noise rather than the general signal

Regularization: penalize very large weight values, because those tend to indicate overfitting

Large-magnitude weights often signal that the network is memorizing idiosyncrasies in the training data

Roughly speaking, uses big weights to find every single wiggle—i.e. fit the noise rather than the general signal

Solution: ***regularization***: add a term to the loss function that penalizes large weights.

Multiple ways of doing this; common one is called *"L2" regularization*: use the *sum of the squared weights*

Regularization: penalize very large weight values, because those tend to indicate overfitting

L2 regularization formally:

$$L = L_0 + \frac{\lambda}{2n} \sum_w w^2$$

where L is the loss function, L_0 was the original loss function, n is the total number of weights, w is the weight, and λ is the *regularization parameter*

So this introduces another *hyperparameter*, λ

Dropout: remove a random subset of the model's neurons each mini-batch of training

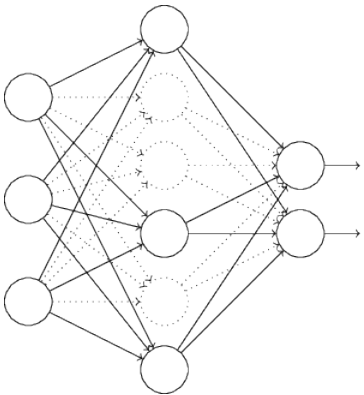

Figure:
http://neuralnetworksanddeeplearning.com/images/tikz31.png

Idea: randomly turning off neurons during training forces the network to rely on multiple redundant pathways
rather than memorizing specific connections

Dropout: remove a random subset of the model's neurons each mini-batch of training

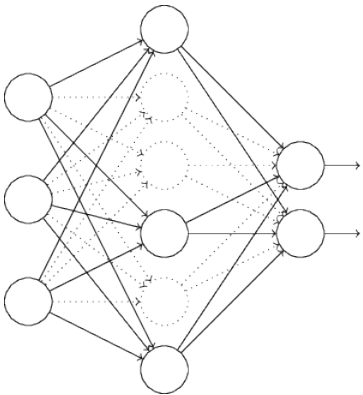

Figure:
http://neuralnetworksanddeeplearning.com/images/tikz31.png

Idea: randomly turning off neurons during training forces the network to rely on multiple redundant pathways
rather than memorizing specific connections

Typically, 1/2 of neurons are turned off each time, but other fractions are possible
Introduces another *hyperparameter*

Data augmentation: get more data, either real or “synthetic”

Recall: virtually all modeling, whether ML-based or not, is about extracting *signal* from the *noise*

Data augmentation: get more data, either real or “synthetic”

Recall: virtually all modeling, whether ML-based or not, is about extracting *signal* from the *noise*

All else equal, more data = more signal but same amount of noise.
Thus: more data = higher SNR (signal-to-noise ratio)

Data augmentation: get more data, either real or “synthetic”

Recall: virtually all modeling, whether ML-based or not, is about extracting *signal* from the *noise*

All else equal, more data = more signal but same amount of noise.
Thus: more data = higher SNR (signal-to-noise ratio)

And recall: overfitting amounts to the model capturing the *noise* (in addition to the signal
So all together: more data = less likelihood of overfitting

Data augmentation: get more data, either real or “synthetic”

How to get more data? Two ways:

Data augmentation: get more data, either real or “synthetic”

How to get more data? Two ways:

1. Real data: go out and get more samples!
   However, sometimes impossible (can’t re-do a field campaign) or just really difficult/expensive.

Data augmentation: get more data, either real or “synthetic”

How to get more data? Two ways:

1. Real data: go out and get more samples!
However, sometimes impossible (can’t re-do a field campaign) or just really difficult/expensive.

2. generate *synthetic data*: E.g. in the handwritten digits example, make copies of each image that are slightly rotated, shifted, and/or distorted. Can use this to easily 2x or more the amount of training data!
In Earth sciences: how to do this is much trickier...but it’s not impossible!

Unit 09: Introduction to Neural Networks

Almost always good idea to *scale* (a.k.a. *normalize*) your training data

For example, via *standardized anomalies* (a.k.a. *z-score normalization*): subtract the mean, divide by the standard deviation: $x_{\text{s.a.}} = (x - \hat{\mu}_x)/\hat{\sigma}_x$

Observe: dimensionless, centered on zero, and with most values $|x_i| \lesssim 1$.

Almost always good idea to *scale* (a.k.a. *normalize*) your training data

For example, via *standardized anomalies* (a.k.a. *z-score normalization*): subtract the mean, divide by the standard deviation: $x_{\text{s.a.}} = (x - \hat{\mu}_x)/\hat{\sigma}_x$

Observe: dimensionless, centered on zero, and with most values $|x_i| \lesssim 1$.

This make training more reliable by putting all predictors on comparable numerical ranges

Prevents any single variable's units or magnitude from dominating the training

Data transformation: can use e.g. log or square-root to make your data more Gaussian, which often helps.

E.g. precipitation: mostly zeros, and non-zero values have very long right tail. This alone makes it harder to model!

Solution: apply a *transformation* to your data. Two common choices are square root and logarithm. Then at the end you apply the inverse transform (square or exponential, respectively), to get back to the actual data.

For log, however, log(0) is not defined, so for precip P you'd need to do e.g. $\log(P + \epsilon)$, where ϵ is some small constant, e.g. 0.01.

For input data, can combine transforms and scaling: e.g. for precip, first transform, then z-score normalize. (In that order!)

Key Takeaways

A feedforward neural network is a differentiable nonlinear function built from layers, with parameters learned by minimizing a scalar loss.

In other words: It's an input layer, an output layer, and zero or more hidden layers of neurons, each with an *activation function*, a *bias*, and, for each neuron in the layer before it, *weights*.

Key Takeaways

A feedforward neural network is a differentiable nonlinear function built from layers, with parameters learned by minimizing a scalar loss.
In other words: It's an input layer, an output layer, and zero or more hidden layers of neurons, each with an *activation function*, a *bias*, and, for each neuron in the layer before it, *weights*.

Stochastic gradient descent—with the gradient of the loss function computed via backpropagation—makes it computationally feasible to train these models on large datasets.

Key Takeaways

A feedforward neural network is a differentiable nonlinear function built from layers, with parameters learned by minimizing a scalar loss.
In other words: It's an input layer, an output layer, and zero or more hidden layers of neurons, each with an *activation function*, a *bias*, and, for each neuron in the layer before it, *weights*.

Stochastic gradient descent—with the gradient of the loss function computed via backpropagation—makes it computationally feasible to train these models on large datasets.

Understanding generalization and overfitting is as important as understanding the mechanics of training: it's all a waste if your model ends up overfitted in the end.