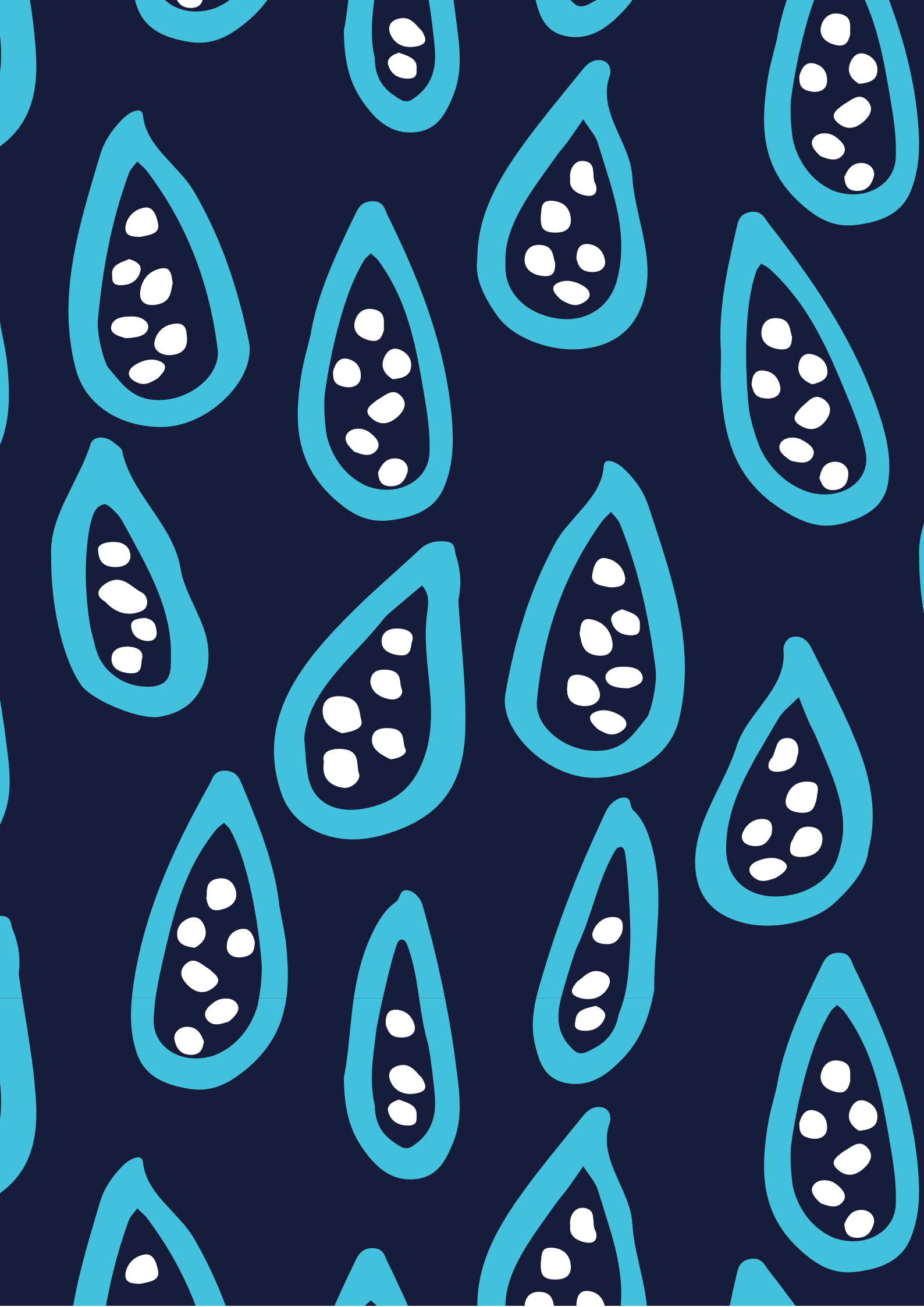


Table of Contents

Machine Learning Basics

- **Feature** - attribute/characteristic of what you are studying
  - Often select, transform, or create new features to improve performance of the model (called feature engineering)
  - Inputs to the model that are useful to "train" it
  - Often encoded in a **feature/design matrix**

- **Classification models** aim to separate classes in the feature space
- **Clustering algorithms** groups "close" data points
- **Regression** looks at relationship between predictors and response

- **Supervised machine learning** - each training example is paired with output **label** (included in dataset)
  - Difficult for large datasets to all be labeled

- **Parameters** like weights / biases are adjusted automatically by algorithm during training, but **hyperparameters** (changes architecture) are determined by programmer manually
  - Different NNs have different parameters; GNN parameters include edge weights and knowing which nodes to connect in graph

- **Epochs** - number of times training dataset is passed through NN during training (means fixed point in time)
- Hyperparameters include number of hidden layers and neurons in them, learning rate, etc.
- **Inference** - another name for testing (when model infers stuff)

- Even though we understand some mathematics of how NN works, it's difficult to interpret what each neuron is doing
  - E.g. for image recognition all it does is minimize a cost function, doesn't actually observe **patterns**

- **Tensor** - multidimensional array

Artificial intelligence - theory/development of computers to do tasks normally requiring human intelligence

↓

machine learning - gives computers ability to "learn" (adjust parameters) without being explicitly programmed

↓

deep learning - machine learning algorithms using neural networks

Underfitting, Overfitting, and Regularization

- **Generalization** - the ability to perform well on previously unseen inputs
  - In machine learning we not only optimize by reducing training error but also test/generalization error

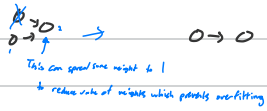
- **Underfitting** - high training and validation error
  - High bias, low variance (bias = error; variance = same errors regardless of training set used)
  - Model is too simple and can't recognize complex features
  - Solution: add complexity to model architecture

- **Overfitting** - training error is low as val validation error is high, indicating memorization/overfitting of training
  - Also known as capturing noise, random variations/errors that do not represent true underlying patterns (e.g. drawing a 1 that looks like a 7
  - Low bias, high variance
  - Features in training recognized very quickly (problem for small datasets)
  - Solution: Use regularization or increase dataset size from data set using data augmentation

- **Capacity** determines how well model can capture patterns in the training data

- **Regularization** - nullifying effect of certain neurons (prevents overfitting)
  - ↑ Complexity in NN ⇒ complex non linearity fit to training data ⇒ overfitting

- **Dropout regularization** - For each batch, set half the neuron activations in each layer to 0; prevents a neuron from being too reliant on output of others



  - Say $\vec{a}^{[L]} = \begin{bmatrix} 0.3 \\ 0.4 \\ 0.5 \end{bmatrix}$. A binary dropout mask of same size as $\vec{a}^{[L]}$ is created by first creating tensor filled with random values between 0 and 1.

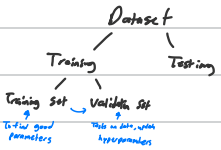
Then, due to Law of Large Numbers, we can create the binary mask by seeing if the result is ≥ keep_rate. Finally do $\vec{a}^{[L]} \odot \vec{d}^{[L]}$ to get final activations of layer L.

- Dropout only used in training & the activations are also scaled in training by $\frac{1}{\text{keep\_rate}} = \frac{1}{1-\text{drop\_rate}}$ so that are similar values in training and inference
  - The less neurons I keep the more I scale their activations

- Dropout is more useful when training over many epochs
  - No dropout ⇒ each neuron relies on precise patterns from all other neurons ⇒ learns very specific features/noise from training data ⇒ overfitting
  - Dropout ⇒ each forward pass uses different subset of neurons ⇒ neuron doesn't rely on very specific neurons/features ⇒ less overfitting
    - BUT too much dropout ⇒ too few neurons in FF ⇒ can't capture patterns in data ⇒ underfitting ⇒ does worse on test set
  - Additional computations can slow down training

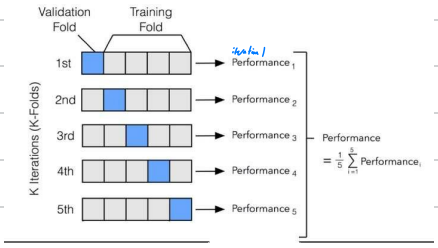
- **L2/ridge regularization** - add term to cost function to penalize larger weights

Cross-Validation



- Validation data estimates generalization error (does same calculations as test set) and updates hyperparameters (usually 80/20 split)
- Validation set is good because otherwise you adjust hyperparameters to make test accuracy better, which may overfit to the test data and not generalize
  - Test data being completely independent of how the model is trained makes it accurate measure of model performance on unseen data (*)
    - If you tune it to test data with same distribution as training in theory that give similar hyperparameters but then you have nothing to evaluate to
    - If you already did hyperparameter tuning with validation accuracy, it is okay to train model multiple times and choose the one that generalizes best (high test accuracy)

- Cross-validation involves splitting training data into K folds/subsets, and using k-1 folds for training and the last one for validating
  - This process is repeated k times with each time having a different fold as validation set (each point in validation one and in training set k-1 times)
  - K-folds cross validation follows this, where on the ith trial, the ith fold is the validation set
  - Each data point used for training and validation eventually
  - Each iteration has a specific (usually same as not) # of epochs before going onto next iteration with different validation set (hyperparameters consistent across different folds)
  - After going through k-folds and seeing error, adjust hyperparameters
- * Every point is eventually in training data, so k-folds better captures distribution of all training data so hyperparameters won't be super different



  - K-folds used to evaluate performance of deep learning model; averaging performance over multiple folds reduces variance of performance cause by randomness of single train-test split
    - to person as good number
- ↑ folds ⇒ ↑ % of train data across all folds ⇒ model trained on higher proportion of data ⇒ model learns better patterns and validation accuracy/model performance averaged over more folds ⇒ accuracy/performance metrics less influenced by any single split/fold (less variance in val accuracy)
  - ? bc more training data on more folds to avg
  - ⇒ hyperparameters are more independent of split used ⇒ better hyperparameter tuning

- ↑ folds ⇒ smaller validation subset that may not be representative of all the data ⇒ more bias
  ↓ folds ⇒ less trials to average ⇒ more variability in performance

- After finding best model with cross-validation, train your new model using all the training data
  - Distribution - refers to spread of features across dataset (i.e. class distribution, brightness, RGB values, etc.)
    - Theoretically (although training/validation split perfectly capturing all training data distribution is rare), if that happen in split for cross-validation, hyperparameters would be independent of split
      - Since this is theoretical, even with cross-validation the best hyperparameters for model trained on full training data could be a little different

- Various ways to find best hyperparameters (with hyperparameter tuning)
  - Grid search - cycle through each combo of hyperparameters and see how well they perform on validation set
  - Chose ones that lead to consistent performance on each fold (would probably handle cotire distribution well)

Properly Creating Train Dataset with Normal and Augmented Examples

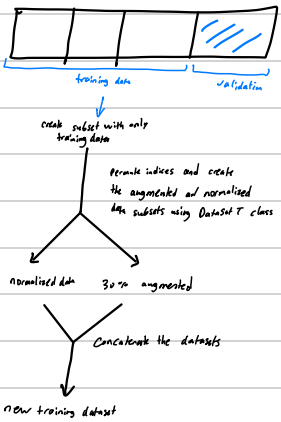
- Can help model differentiate between various classes
- Shown below are two approaches I came up with to create a new dataset that contains both normal and augmented data
    - One reason this was done is because more training data resulted in a better model
- These approaches to create the new dataset for training were used in both k-folds cross validation and the final training of the model
- Importantly, neither of these approaches augment examples of the validation set during k-folds cross validation

- The visualizations of each approach use the following split: 70% normalized images, 30% augmented

- Approach 1 - uses custom DatasetT class to define transformations per image

K-folds:

Untransformed Train/Val Data



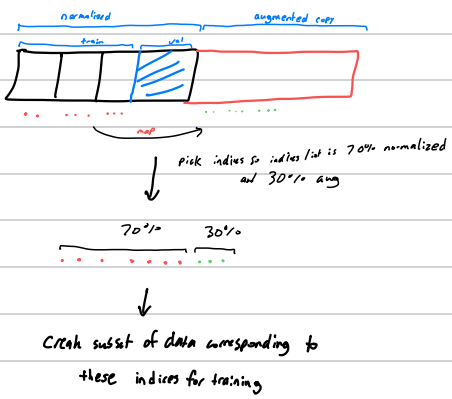
- For full training data, the same applies except there is no validation set

- Approach 2

K-folds: Each fold...

- Generate train and val indices on the normalized dataset
- Concatenate an augmented dataset same size as normalized
- Map the train indices to the augmented copy
- Choose 70% of all original train indices to be from normalized and the other 30% from augmented
- Create a subset of data with these indices; this dataset is the same size as original normalized train subset

Concatenated:



Create subset of data corresponding to these indices for training

- For final training with all training data, use similar approach except...
  - To maximize number of training examples in dataset, use 100% of the normalized data and an additional amount of augmented data so the proportion is still 70% normal, 30% augmented

$$(\text{total training examples}) \cdot (\text{prop\_normal}) = \text{len}(\text{normalized\_data}) \Rightarrow \text{total training examples} = \frac{\text{len}(\text{normalized\_data})}{\text{prop\_normal}}$$

- Additionally, for the augmented data, after analyzing confusion matrix, I replaced some examples of classes the model performed well on with classes the model performed poorly on

Approach 2 performed better, and I believe it did due to the following:
- Mapping the indices onto a new dataset provided greater similarity between the normal(ized) and augmented training examples
  - These similar examples in Approach 2 train dataset could have made the model learn better since it is seeing more similar data as opposed to an augmentation of an image it has never seen in the batch before (as in Approach 1)
- Also improved performance in final training due to more training data

The Perceptron Algorithm

- It's a supervised binary classification model whose goal is to find the hyperplane that separates the two classes of data points in a feature space
  - Used when a dataset is linearly separable (fails XOR problem)
  - Single layer neural network (parameters that are adjusted in the algorithm are weights and biases)
  - Called "perceptron" because the model stimulates human perception and perceptron perceives data

$0 = \vec{w} \cdot \vec{x} + b$ is hyperplane

$y = \vec{w} \cdot \vec{x} + b$ ← bias

(output, weight, input feature vector)

$w_1 x_1 + w_2 x_2 \dots$

$\vec{w} = \begin{bmatrix} \vec{w}_1 \\ \vec{w}_n \end{bmatrix} \quad \vec{x} = \begin{bmatrix} x_1 \\ x_n \end{bmatrix}$

- If you have n features/dimensions, you have n components in $\vec{w}$ and $\vec{x}$
  - The weight components adjust hyperplane orientation/slope

- **Step function** - makes input abruptly "step" from one value to another at a specified threshold (a piecewise function)

- This perceptron contains...
  - An input layer, where each neuron contains a feature of the data point fed in
  - An output layer, consisting of one neuron with the binary input

For each data point i...

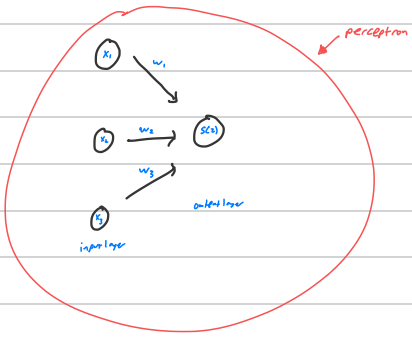
- Calculates the weighted sum $\vec{w} \cdot \vec{b}$ → adds the bias → passes result through step function (the specific activation function) → updates $\vec{w}$ and $\vec{b}$ when perceptron misclassifies the data → finish when $\hat{y}_i = y_i$ for all training examples (predicted, target vector; in bounds; i.e. the algorithm converges)
- Step function: $\hat{y} = \begin{cases} 1 & \vec{w} \cdot \vec{x} + b > 0 \\ 0 & \vec{w} \cdot \vec{x} + b \leq 0 \end{cases}$ (binary classes; z)
- Updating parameters

$$\vec{w} \leftarrow \vec{w} + \eta (y - \hat{y}) \vec{x}$$

$$b \leftarrow b + \eta (y - \hat{y})$$

(assigning; learning rate)

S = step, z = weighted sum $= \sum_{i=0}^{n} w_i x_i$



- Cannot separate nonlinear classes or solve XOR problem, multiple perceptrons needed with non-linear activation
  - With perceptron algorithm the hyperplane cannot separate the classes

Parameter Initialization

- **He/Kaiming initialization**
  - Technique used to randomly initialize parameters when ReLU is used
  - Weights are initialized as random numbers from Gaussian/normal distribution (then multiplied by scaling factor)
    - Mean at 0 ⇒ center of bellcurve at 0 ⇒ most weights are zero and are small positive and negative
      - SD is spread of curve
    - Scaling factor - $\sqrt{\frac{2}{\text{input\_dim}}}$ ← normal distribution with ?
    - Scaling factor helps set variance of weights to proper value and prevent vanishing/exploding gradients

- We also want inputs to be normalized because larger inputs ⇒ numerical instability and $\frac{d}{dz}[\text{activation}(z)]$ is extremely small ⇒ vanishing gradient problem

Multilayer Perceptrons and Forward Propagation

- **Convolutional neural network** - good for image recognition
- **Long short-term memory network** - good for speech recognition
- **MNIST dataset** - popular dataset of handwritten digits 0 through 9 in black and white
- Neural networks have input layer, hidden layers, and output layer
  - For image recognition it is common for each neuron of the inner layer to represent a pixel attribute (e.g. brightness/color)
  - A **neuron** is like $f(\vec{x}, \vec{w}) = \sigma(\underbrace{\sum x_i w_i}_{z} + b)$ ($\vec{x}, \vec{w}$: weights of previous layer; σ: activation function)
- The "basic" neural network is a **multilayer perceptron** (MLP)
- **Learning** - finding the right parameters
- For classification, neuron in output layer with highest activation is the predicted class
- Superscripts indicate the layer; subscripts indicate the neuron in the layer
  - All connections have a weight, and there is 1 bias in each neuron

- $z_j^\ell = \left(\sum_i w_{ij}^\ell a_i^{\ell-1}\right) + b_j^\ell$
  - z_j^ℓ is the weighted sum for neuron j in layer ℓ
  - w_{ij}^ℓ is the weight for connection from neuron i in layer ℓ to neuron j in layer $\ell-1$ (seems backward)
  - b_j^ℓ is bias for neuron j in layer ℓ ($j=0$ is 1st neuron)
- $a_j^\ell = \sigma(z_j^\ell)$ (a: activation)
- 0 is output layer, h is hidden layer
- Feed forward calculated more efficiently with linear algebra

Shorthand for $a_j^\ell = \sigma(\underbrace{\sum_i w_{ij}^\ell a_i^{\ell-1}}_{} + b_j^\ell)$

$$W = \begin{bmatrix} w_{00} & w_{01} & \cdots & w_{0n} \\ \vdots & & & \\ w_{n0} & w_{n1} & \cdots & w_{nn} \end{bmatrix} \qquad \vec{a}^{\ell-1} = \begin{bmatrix} a_0^{\ell-1} \\ \vdots \\ a_n^{\ell-1} \end{bmatrix}$$

(W: matrix of connections between the layers)

- W_k is all connections from layer $\ell-1$ to neuron i in layer ℓ
- $W_k \vec{a}_j = \vec{w}_k^T \cdot \vec{a}_j = z_j$, the weighted sum for that neuron

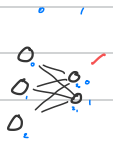
$$\vec{z}^\ell = \underbrace{W_i}_{\text{between layer } \ell \text{ and } \ell-1 \text{ (weight connections between layers)}} \vec{a}^0 + \vec{b} = \begin{bmatrix} w_{00} & w_{01} & \cdots & w_{0n} \\ \vdots & & & \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{bmatrix} \begin{bmatrix} a_0^0 \\ \vdots \\ a_n^0 \end{bmatrix} + \begin{bmatrix} b_0 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} \sum_i w_{i0}^\ell a_i^0 \\ \vdots \\ \sum_i w_{in}^\ell a_i^0 \end{bmatrix} + \begin{bmatrix} b_0 \\ \vdots \\ b_n \end{bmatrix} = \underbrace{\begin{bmatrix} z_0 \\ \vdots \\ z_n \end{bmatrix}}_{\text{weighted sum + bias for neuron 0 in layer } \ell} \qquad \sigma(\vec{z}) = \vec{a}_\ell = \begin{bmatrix} a_0^\ell \\ \vdots \\ a_n^\ell \end{bmatrix}$$

- Each component of the matrix vector product is a weighted sum for that corresponding neuron in layer ℓ

- There is a weight matrix and a bias vector for each layer of NN
- $\vec{z}^\ell$ or $\vec{a}^\ell$ usually matrix (each row is outputs of an example; # of rows is batch size): $Z = AW + \vec{b}$ (really adding this $\vec{b}$ to each row)
  - for parallelism
  - batch size x input
  - input x output

Analyzing the Forward Propagation Process

Between Layer 0 and 1



$A_0 = X$

$z_0^1 = w_{00}x_0 + w_{01}x_1 + w_{02}x_2 = \begin{bmatrix} w_{00} \\ w_{01} \\ w_{02} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w_{00} & w_{01} & w_{02} \end{bmatrix} \vec{x}$

$z_1^1 = w_{10}x_0 + w_{11}x_1 + w_{12}x_2 = \begin{bmatrix} w_{10} & w_{11} & w_{12} \end{bmatrix} \vec{x}$

$W_1 X + B_1 = Z$ (features × batch size)

$\vec{z}^1 = \begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = W\vec{x}$

(number of features in)

$z_i^1 = w_i^1 \vec{x} = w_i^{1\,T} \cdot \vec{x}$

(ith row)

- ith row of matrix W contains weight of connections from ith neuron in layer ℓ to all neurons in layer $\ell-1$

- For batch size > 1 there are multiple $\vec{x}$'s with same calculation, which corresponds to $X = \begin{bmatrix} \vec{x}_1^1 & \vec{x}_2^1 \end{bmatrix}$ (feature of one example)
  - More generally for connections between layer $\ell-1$ and layer ℓ, $A^{\ell-1} = \begin{bmatrix} \vec{a}_1^{\ell-1} & \cdots & \vec{a}_b^{\ell-1} \end{bmatrix}$ (batch size)

(each row corresponds to diff neuron, each column is all activations in layer $\ell-1$ of certain example)

$$\begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \end{bmatrix}^{\ell} \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \\ a_{20} & a_{21} \end{bmatrix}^{\ell-1} + \begin{bmatrix} b_0 & b_0 \\ b_1 & b_1 \end{bmatrix}^{\ell} = \begin{bmatrix} z_{00} & z_{01} \\ z_{10} & z_{11} \end{bmatrix}^{\ell}$$

(columns are same since biases of neurons don't change per example; outputs in layer ℓ for first example)

$\sigma(z^\ell) = A^\ell$ (activation vector)

The Weight Matrix - Fundamental Subspaces

- Basis vector j is all connections from neurons ℓ to neuron $j^{\ell-1}$

- $C(W)$ contains all possible outputs of $W\vec{x} = \vec{z}$ before bias and activation
  - Orthogonal initialization for W columns ensures network can map to larger, more varied set of inputs ⇒ can fit complex data earlier

- $N(W)$ shows what layers of activations give $\vec{z} = 0$ (which directions of input space are "ignored")
  - High dimensional $N(W)$ ⇒ much of input space being ignored

- $R(W)$ related to how much all the features matter to each neuron j^{ℓ}; linearly independent rows ⇒ each neuron has different weights for features/inputs ⇒ each neuron contributes more info
- $N(W^T)$ less interpretable

- Optimizing $C(W)$ and $R(W)$ can lead to simpler, effective model

- W^T used for backpropagation

Cross-Entropy Loss

- Viewing why cross entropy is a good loss function for classification

e.g. Good prediction

	$\hat{y}$ = softmax(logits)	y
Class 0	0.99	1
Class 1	0.01	0

$$-\sum_{i}^{N} y_i \log(\hat{y}_i) = -((1)\log(.99) + (0)\log(0.01)) = -\log_{10}(0.99) = 0.004$$ ← low loss for good classification

e.g. Bad Prediction

	$\hat{y}$ = softmax(logits)	y
Class 0	0.01	1
Class 1	0.99	0

$$-\sum_{i}^{N} y_i \log(\hat{y}_i) = -((1)\log(.01) + (0)\log(0.99)) = -\log_{10}(0.01) = -(-2) = 2$$ ← high loss for bad classification

- In ML, $\log_e$ is commonly used for its simple derivative, but the same properties as above apply

Backpropagation

← Backward propagation of errors

- **Backpropagation**: the algorithm for determining how a single training example would like to nudge the weights and biases in terms of what changes to these parameters cause the most rapid decrease to the loss (calculates gradient)

- Based on output you want, you know how to adjust the parameters in the layer before it
  - Prioritize adjusting the neurons whose activation is not close to what you want

To increase activation of neuron j in layer ℓ...

- Increase b_j
- Increase $w_{j,i}$ in proportion to a_i
  - $w_{j,i}$: j = ℓ, i = $\ell-1$
- Increase a_i in proportion to $w_{j,i}$:
  - Higher activations and weights complement each other in changing the output since $\uparrow a_i$ for higher $w_{j,i} \Rightarrow$ higher $a(z)$

- We can start which of these actions decreases the cost the most (most efficient step):
  - Tiny change in w_i changes loss the most $\Rightarrow \left|\frac{\partial L}{\partial w_i}\right|$ = High $\Rightarrow w_i - \eta \frac{\partial L}{\partial w_i}$ causes a bigger step (one reason why we use derivative for gradient descent)

- Doing this can "strengthen connections" / make it more likely for that output neuron to be activated for specific training example

- Weights can be negative
  - Basically to make O^{ℓ} (layer ℓ) brighter you can decrease activations for the negative weights and increase them for positive ones to have greatest effect on L
  - But you have to take all O into account and how each of them want to change a_i (really just adjusting weights and biases)
    - And all these "wanted" changes are added together
    - This is what propagating backwards is/gives you list of nudges to apply to layer $\ell-1$

- Each training example will have how it wants to nudge each weight and bias, so take the average
  - This nudge is proportional to negative gradient (i.e. larger nudge means ⇑ larger partial derivative)

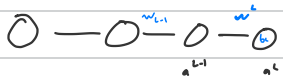
$$\theta_i \leftarrow \theta_i - \eta \frac{\partial L}{\partial \theta_i} \Rightarrow \theta_i \leftarrow \theta_i + \Delta\theta_i \Rightarrow \Delta\theta_i = -\eta \frac{\partial L}{\partial \theta_i} \qquad \frac{\Delta\theta_i}{\eta} = -\frac{\partial C}{\partial \theta_i}$$

↑ gradient ⇒ steeper slope ⇒ larger change

Using the chain rule to find the gradient

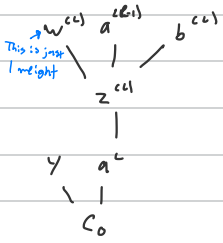
- Start by propagating/transmit the loss backwards throughout the network; it computes the partial derivatives layer by layer

- Simple NN: $C(w_1, b_1, \dots, w_3, b_3)$ Loss function: MSE



$C_0(\dots) = (a^{(L)} - y)^2$ $\quad a^L = \sigma(z^L)$ $\quad z^L = w^L a^{L-1} + b^L$

$C_0(\dots)$: cost for single training example; final cost is avg of C_k — y: target activation



$\partial w \to \partial z \to \partial a \to \partial w$, so we should see how they change in relation to z and to on until C_0 (chain rule)

- We want to see how sensitive cost function is to weight $\frac{\partial C_0}{\partial w^L}$ (nudging a weight to cost)

for specific training example

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^L}{\partial w^L} \cdot \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L}$$

- Chain rule is much more efficient than plugging in $\sum_i w_j a_i + b$ backwards into each a_i for network to derive cost function then taking the partials
  - Chain rule allows for reusing your previous computation, and it is recursive so it can be written more easily as an algorithm

$$\frac{\partial C}{\partial w^L} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^L}$$

$\frac{\partial C}{\partial w^L}$: 1 component of ∇C — $\frac{\partial C_k}{\partial w^L}$: 1 training example; need to average over all training examples

$\frac{\partial C_0}{\partial a^L} = 2(a^L - y)$ Cost is proportional to error $\quad \frac{\partial a^L}{\partial z^L} = \frac{da^L}{dz^L} = \sigma'(z^L)$ $\quad \frac{\partial z^L}{w^L} = a^{L-1}$ amount the weight changes z is dependent on a^{L-1}

$$\frac{\partial C_0}{\partial w^L} = a^{(L-1)} \sigma'(z^L) 2(a^L - y)$$

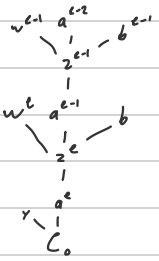
$$\frac{\partial C_0}{\partial b^L} = \frac{\partial z^L}{\partial b^L} \cdot \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L} = \sigma'(z^L) 2(a^L - y)$$

$\frac{\partial z^L}{\partial b^L}$: 1

$$\frac{\partial C_0}{a^{L-1}} = \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial C_0}{\partial a^L} = a^{(L-1)} \sigma'(z^L) 2(a^L - y)$$

Propagating Backwards

Going Further Back in Network



$z^l = w^l a^{l-1} + b^l \Rightarrow z^{l-1} = w^{l-1} a^{l-2} + b^{l-1} \Rightarrow \frac{\partial z^{l-1}}{\partial w^{l-1}} = a^{l-2}$ or $\partial z^{(l-1)} = a^{(l-2)} \partial w^{l-1}$

↑ makes sense for single neuron layers

⋮

and so on

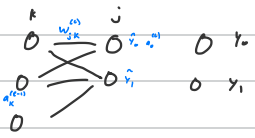
$$\frac{\partial C_0}{\partial w^{(l-1)}} = \frac{\partial z^{l-1}}{\partial w^{l-1}} \cdot \frac{\partial a^{l-1}}{\partial z^{l-1}} \cdot \frac{\partial z^l}{\partial a^{l-1}} \cdot \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial C_0}{\partial a^l} = \dots$$

Process

Replace l with $l-1$ in your equation → find unknown partial → take its average across all training examples in mini-batch → use formula/gradient descent to adjust parameters

With Indices

n_l - number of neurons in layer l



$$C_0 = \sum_{j=0}^{n_l - 1} (a_j^{(l)} - y_j)^2 \qquad z_j^l = \sum_{k=0}^{n_{l-1}} w_{jk}^l a_k^{l-1} + b_j^l \qquad a_j^l = \sigma(z_j^l)$$

- Weight indices feel less backwards for backprop

$$\frac{\partial C_0}{w_{jk}^l} = \frac{\partial z_j^{(l)}}{w_{jk}^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot \frac{\partial C_0}{\partial a_j^l}$$

↑ follow "path" from that weight to the cost

Same as before:

$$\frac{\partial z_j^l}{w_{jk}^l} = a_k^{l-1} \qquad \frac{\partial a_j^l}{\partial z_j^l} = \sigma'(z^l{}_j)$$

Different:

$\frac{\partial C_0}{a_k^{(l-1)}}$: the activation of a neuron in another layer has multiple paths to output neurons, so you have to trace each of these outputs and how they each affect the cost function

↑ This has multiple paths, so you have to sum of partial for each path to get total effect of a_k^{l-1} on C_0

$$\frac{\partial C_0}{\partial a_k^{l-1}} = \sum_{j=0}^{n_l - 1} \frac{\partial z_j^l}{\partial a_k^{l-1}} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot \frac{\partial C_0}{\partial a_j^l}$$

Sum partials to account for total amount that ○ affects $z \to a \to$ cost

$$\frac{\partial C_0}{\partial a_j^L} = 2(a_j^L - y_j) \quad \text{or} \quad \sum_{j=0}^{n_{l+1}} w_{jk}^{l+1} \sigma'(z_j^{l+1}) \frac{\partial C}{\partial a_j^{(l+1)}}$$

$\frac{\partial z_j^{l+1}}{\partial a_j^l}$ · $\frac{\partial a^{l+1}}{\partial z_j^{l+1}}$ solved

summing up all paths from activation neuron

recursion until end with $2(a_j^L - y_j)$

$$\frac{\partial C_0}{\partial w_{jk}^{(l-1)}} = \frac{\partial z_j^{l-1}}{\partial w_{jk}^{l-1}} \cdot \frac{\partial a_j^{l-1}}{\partial z_j^{l-1}} \cdot \frac{\partial z_j^l}{\partial a_j^{l-1}} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot \frac{\partial C_0}{\partial a_j^l}$$

δ_j^l

$\frac{\partial C_0}{\partial a_j^{l-1}}$, already calculated

- Same idea of summing the paths until the end
- $\delta_j^l = \frac{\partial C_0}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_j^l}$ is defined as the **error** of neuron j in layer l; determines how much that neuron corresponds to the error in output

↑ Summation like before

Backpropagation Derivations

- Option 1: Calculate loss for each training example ⇒ calculate DC_i for each training example ⇒ gradient is $\frac{1}{m}\sum_i C_i$ ← more common, less computationally expensive
- Option 2: Calculate loss for each training example and average it with $-\frac{1}{m}\sum_{\text{all}}\sum_i y_i \log(\hat{a}_i)$ ⇒ Find partial derivatives with this avg Loss

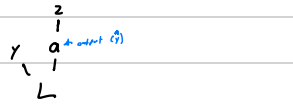
Backpropagating Error from Loss → Final Layer

(includes parameters that give the output of this layer)

Loss Function: Softmax categorical cross entropy

$$\text{Loss} = -\sum_{\text{all}} y_k \log(a_k) \qquad a_k = \frac{e^{z_i}}{\sum_{k=k} e^{z_k}} = \frac{e^{z_i}}{e^{z_i} + e^{z_j} + \sum_{k\neq i,j} e^{z_k}}$$

(a_k: probability; z_i: logit)



$$\frac{\partial L}{\partial \vec{z}} = \frac{\partial L}{\partial \vec{a}} \cdot \frac{\partial \vec{a}}{\partial \vec{z}} = ?$$ Single example

$$\frac{\partial \vec{a}}{\partial \vec{z}} = \frac{\partial}{\partial \vec{z}}\vec{a} = \begin{bmatrix} \frac{\partial a_1}{\partial z_1} & \frac{\partial a_1}{\partial z_2} & \cdots \\ \frac{\partial a_2}{\partial z_1} & \frac{\partial a_2}{\partial z_2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

$\frac{\partial a_i}{\partial z_j}$ ← individual neuron in last layer

$j \neq i$

$$\frac{\partial}{\partial z_j} a_i = \frac{\partial}{\partial z_j}\left(\frac{e^{z_i}}{e^{z_j} + \sum_{k\neq j} e^{z_k}}\right) = \frac{-e^{z_i}\left(e^{z_j}\right)}{\left(e^{z_j} + \sum_{k\neq j} e^{z_k}\right)^2} = \frac{-e^{z_i}\left(e^{z_j}\right)}{\left(\sum e^{z_k}\right)^2} = \frac{-e^{z_i}}{\sum e^{z_k}} \cdot \frac{e^{z_j}}{e^{z_k}} = -a_i a_j$$

even with $i \neq j$ and usually a_i depends only on z_i, softmax activation uses all components of $\vec{z}$

$j = i$

$$\frac{\partial a_i}{\partial z_i} = \frac{\partial}{\partial z_i} a_i = \frac{\partial}{\partial z_i}\left(\frac{e^{z_i}}{e^{z_i} + \sum_{k\neq i} e^{z_k}}\right) = \frac{e^{z_i}\left(e^{z_i} + \sum_{k\neq i} e^{z_k}\right) - e^{z_i}\left(e^{z_i}\right)}{\left(\sum e^{z_k}\right)^2} = \frac{e^{z_i^2} - e^{z_i^2} + e^{z_i}\sum_{k\neq i} e^{z_k}}{\left(\sum e^{z_k}\right)^2} = \frac{e^{z_i}}{\sum e^{z_k}} \cdot \frac{\sum_{k\neq i} e^{z_k}}{\sum e^{z_k}} = a_i(1-a_i)$$

$$\frac{\sum_{k\neq i} e^{z_k}}{\sum e^{z_k}} = \frac{\sum e^{z_k} - e^{z_i}}{\sum e^{z_k}} = 1 - \frac{e^{z_i}}{\sum e^{z_k}} = 1 - a_i$$

$$\frac{\partial a_i}{\partial z_j} = \begin{cases} -a_i a_j & i \neq j \\ a_i(1-a_i) & i = j \end{cases}$$

$$\text{Loss} = -\sum y_k \log(a_k) = -\left[y_i \log(a_i) + \sum_{k\neq i} y_k \log(a_k)\right]$$

$$\frac{\partial L}{\partial z_i} = \frac{\partial a_i}{\partial z_i}\left(\frac{\partial}{\partial a_i}[L]\right) = \frac{\partial a_i}{\partial z_i}\left(\frac{\partial}{\partial a_i}\left[y_i \log(a_i) + \sum_{k\neq i} y_k \log(a_k)\right]\right) = \frac{\partial a_i}{\partial z_i}\left(-\frac{y_i}{a_i} - \sum_{k\neq i}\frac{y_k}{a_k}\right) = -\frac{\partial a_i}{\partial z_i}\cdot\frac{y_i}{a_i} - \frac{\partial a_i}{\partial z_i}\cdot\sum_{k\neq i}\frac{y_k}{a_k} = -\frac{\partial a_i}{\partial z_i}\cdot\frac{y_i}{a_i} - \sum_{k\neq i}\frac{\partial a_k}{\partial z_i}\cdot\frac{y_k}{a_k}$$

(inside: $y_i \log(a_i)$ + $\sum_{k\neq i} y_k \log(a_k)$)

$$= \frac{-a_i(1-a_i)\,y_i}{a_i} - \sum_{k\neq i}\frac{-a_k a_i\, y_k}{a_k} = -(1-a_i)y_i + \sum_{k\neq i} a_i y_k = -y_i + a_i y_i + a_i\sum_{k\neq i} y_k = -y_i + a_i\left(y_i + \sum_{k\neq i} y_k\right) = -y_i + a_i\left(\sum y_k\right) = -y_i + a_i = a_i - y_i$$

($\sum y_k$ is one hot)

$$\delta_i^L = \frac{\partial L}{\partial z_i} = a_i - y_i \Rightarrow \frac{\partial L}{\partial \vec{z}_L} = \delta^L = \vec{a}_L - \vec{y} \quad \text{or} \quad dZ_2 = A_2 - Y \text{ for batches}$$

($\vec{z}_L$: output layer; z_i: one output neuron)

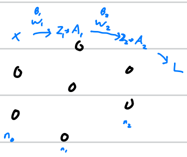
Explanation & further steps below → $\vec{\delta}^L = \nabla_a L \odot \sigma'(z^L) = \begin{bmatrix} \frac{\partial L}{\partial a_1}\cdot\frac{\partial a_1}{\partial z_1} \\ \vdots \\ \frac{\partial L}{\partial a_n}\cdot\frac{\partial a_n}{\partial z_n} \end{bmatrix}$

- Tips for intuition
  - Just as $W\vec{a}_i$ maps info from layer $\ell-1$ to ℓ, $W^T\vec{y}$ maps error backwards from layer ℓ to $\ell-1$
    - This is because $W = j\begin{bmatrix} k \end{bmatrix}$ and $W^T = k\begin{bmatrix} j \end{bmatrix}$, $k \in \mathbb{R}^{|\ell-1|}$, $j \in \mathbb{R}^{|\ell|}$ ← num. neurons in layer ℓ
  - Some formulas have A^T and $\odot$ both so matrix shape outputted is as expected and so the derivatives are scaled properly
    - E.g.

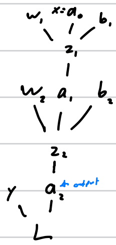
      $f'(z) = \frac{\partial a_i}{\partial z_i}$ is low ⇒ neuron activation not that sensitive to input ⇒ not as important to take steps in weight direction with this derivative ⇒ element-wise multiplication of $\frac{\partial L}{\partial \vec{a}} \odot \frac{\partial}{\partial \vec{z}}$ will be for corresponding derivatives ⇒ smaller steps in certain direction
    - Hadamard product also correctly applies chain rule with vectors/matrices

- For NN with 1 hidden layer

e.g.



=



- For batches use matrices and scale down the summed loss with $\frac{1}{m}$ for W and B_i don't scale down $\frac{\partial Cost}{\partial z}$ since it is intermediary

Activation function $= f$ $\quad Z_i = W_i A_{i-1} + B_i \quad A_0 = X$

$$\frac{\partial Z_i}{\partial W_i} = \frac{\partial}{\partial W_i}\left[W_i A_{i-1} + B_i\right] = A_{i-1} \qquad \frac{\partial Z_i}{\partial B_i} = 1$$

By previous page, $\frac{\partial L}{\partial z_i} = \vec{a}_i - \vec{y}_i$ for single example for last layer ⇒ $\frac{\partial L}{\partial z_2} = \vec{a}_2 - \vec{y}$

$$\underbrace{\frac{\partial L}{\partial W_2}}_{n_2 \times n_1} = \overbrace{\frac{\partial L}{\partial \vec{a}_2} \cdot \frac{\partial \vec{a}_2}{\partial \vec{z}_2}}^{dZ_2} \cdot \frac{\partial \vec{z}_2}{\partial W_2} = \frac{\partial L}{\partial \vec{z}_2} \cdot \frac{\partial \vec{z}_2}{\partial W_2} = \underbrace{(\vec{a}_2 - \vec{y})}_{(n_2 \times 1)} \underbrace{\vec{a}_1^T}_{(1 \times n_1)}$$

(subscript 2 ↑ denotes layer 2 of network; ← 1 example; ↑ must transpose)

For a batch, $$\underbrace{\frac{\partial Cost}{\partial W_2}}_{(n_2 \times n_1)} = \frac{1}{m} \underbrace{\underbrace{(A_2 - Y)}_{(n_2 \times m)} \underbrace{A_1^T}_{(m \times n_1)}}_{n_2 \times n_1}$$

$$\frac{\partial L}{\partial \vec{b}_2} = \overbrace{\frac{\partial L}{\partial \vec{a}_2} \cdot \frac{\partial \vec{a}_2}{\partial \vec{z}_2}}^{d\vec{z}_2 \text{ or } \delta^2} \cdot \frac{\partial \vec{z}_2}{\partial \vec{b}_2} = d\vec{z}_2 \cdot \frac{\partial \vec{z}_2}{\partial \vec{b}_2} = \overbrace{\frac{\partial L}{\partial \vec{z}_2}}^{d\vec{z}_2} \cdot \frac{\partial \vec{z}_2}{\partial \vec{b}_2} = d\vec{z}_2 = \vec{a}_2 - \vec{y}_2$$

↙ 1 training example

✗ $\underbrace{\frac{\partial cost}{\partial B_2}}_{(n_2, 1)} = \frac{1}{m} \underbrace{dZ_2}_{(n_2, m)}$ $\quad$ different sample ⇓ different a ⇓ different dB $\quad \left[dB_{example_1}, \ldots dB_{example_m} \right]$ avg columns

✓ $\underbrace{\frac{\partial cost}{\partial B_2}}_{n_2 \times 1} = \frac{1}{m} \underbrace{Sum(dZ_2, 1)}_{n_2 \times 1}$ ← sum along dimension 1 (2nd dimension - columns); sums all columns and averages them

- We can sum and take avg to find average $d\vec{b}$ across batch and get correct size

Why $\frac{1}{m}$?

- $A^{(2)}A^{(1)T} \Rightarrow a_{ij} = A_i^{(2)T}\vec{a}_j = A_i^{(2)} \cdot \vec{a}_j$. Basically any matrix AX^T involves summing with all columns (samples of A), so we should balance this out with $\frac{1}{m}$

  sample 1 ... sample n: $\begin{bmatrix} a_{21} & a_{21} & a_{21} \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ (sum, sum)

Output layer → Hidden layer

$$d\vec{z}_1^{[1]} = \vec{\delta}^{[1]} = \frac{\partial L}{\partial \vec{z}_1^{[1]}} = \frac{\partial L}{\partial a_1^{[2]}} \cdot \frac{\partial a_1^{[2]}}{\partial z_1^{[2]}} \cdot \frac{\partial z_1^{[2]}}{\partial a_1^{[1]}} \cdot \frac{\partial a_1^{[1]}}{\partial z_1^{[1]}} = \frac{\partial L}{\partial z_1^{[2]}} \cdot w_2 \cdot f'(\vec{z}_1)$$

($\frac{\partial L}{\partial z_1^{[2]}}$: layer 1 neuron 1; $\frac{\partial L}{\partial a_1^{[2]}}$: $(n_2, 1)$; w_2: (n_2, n_1); $f'(\vec{z}_1)$: relu; $\vec{\delta}^{[1]}$: $(n_1, 1)$)

Single neuron in single example

- This works for a single $\frac{\partial L}{\partial z_j^{[l]}}$; for $\frac{\partial L}{\partial \vec{z}^{[l]}}$ (or $Z^{[l]}$, a matrix) (a vector of neurons), we need to adjust the formula to properly apply the chain rule

$$d\vec{z}_1 = \vec{\delta}^{[1]} = \frac{\partial L}{\partial \vec{z}_1} = \frac{\partial L}{\partial \vec{a}_2} \cdot \frac{\partial \vec{a}_2}{\partial \vec{z}_2} \cdot \frac{\partial \vec{z}_2}{\partial \vec{a}_1} \cdot \frac{\partial \vec{a}_1}{\partial \vec{z}_1} = d\vec{z}_2 \cdot w_2 \cdot f'(\vec{z}_1) \quad \times$$

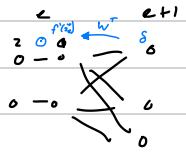
($d\vec{z}_1$: $(n_1, 1)$; $d\vec{z}_2$: $(n_2, 1)$; w_2: (n_2, n_1); $f'(\vec{z}_1)$: relu, $(n_1, 1)$)

adjust for proper matrix size:

$$= (w_2^T d\vec{z}_2) \odot f'(\vec{z}_1) \quad \checkmark$$

(w_2^T: (n_1, n_2); $d\vec{z}_2$: $(n_2, 1)$; $f'(\vec{z}_1)$: $(n_1, 1)$; $w_2^T d\vec{z}_2$: $(n_1, 1)$; $(n_1, 1)$)

- Generally, $\vec{\delta}^{[l]} = \frac{\partial L}{\partial \vec{z}_l} = d\vec{z}^{[l]} = W_{l+1}^T \vec{\delta}^{[l+1]} \odot f'(\vec{z}_l)$ (layer l; $d\vec{z}^{[l+1]}$)



- Hadamard product merely applies chain rule (no matrix multiplication since $\vec{z}$ and $\vec{a}$ are not fully connected) after W^T maps the error backwards

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \vec{a}_2} \cdot \frac{\partial \vec{a}_2}{\partial \vec{z}_2} \cdot \frac{\partial \vec{z}_2}{\partial \vec{a}_1} \cdot \frac{\partial \vec{a}_1}{\partial \vec{z}_1} \cdot \frac{\partial \vec{z}_1}{\partial w} = \frac{\partial L}{\partial \vec{z}_1} \cdot \frac{\partial \vec{z}_1}{\partial w} = (w_2^T d\vec{z}_2) \odot f'(\vec{z}_1) \cdot a_0^T \quad \text{or} \quad d\vec{z}_1 a_0^T$$

($\frac{\partial L}{\partial w_1}$: (n_1, n_0); $d\vec{z}_1$: $\vec{\delta}^2$; $(w_2^T d\vec{z}_2)$: $(n_1, 1)$; $f'(\vec{z}_1)$: $(n_1, 1)$; a_0^T: $(1, n_0)$; (n_1, n_0))

Single example

$$\frac{\partial Cost}{\partial w_1} = \frac{1}{m} d\vec{z}_1 A_0^T$$ (A_0^T: x^T)

Batch

- Generally, $\frac{\partial C}{\partial w^{[l]}} = \vec{\delta}^{[l]} (\vec{a}^{[l-1]})^T$

$$\frac{\partial L}{\partial \vec{b}_1} = \frac{\partial L}{\partial \vec{z}_1} \cdot \frac{\partial \vec{z}_1}{\partial \vec{b}_1} = \frac{\partial L}{\partial \vec{z}_1} = d\vec{z}_1$$

Generally, $\frac{\partial C}{\partial b^{[l]}} = \vec{\delta}^{[l]}$

$$\frac{\partial Cost}{\partial \vec{b}_1} = \frac{1}{m} \text{sum}(d\vec{z}_1, 1)$$

- No need to backpropagate error to input layer

Optimizing the Loss Function with Gradient Descent

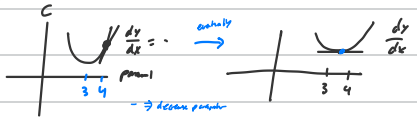
- Gradient descent is how the model knows the amount it should update its parameters

- Take average of $\Sigma(y-\hat{y})^2$ (each output neuron) for all training data to get **loss/cost** L MSE
  - Output of cost function dependent on all the parameters of neural network (hyperparameters aren't direct input)

- Computing minimum is computationally expensive
  - For each step, find the minimum on that step (using single variable calc) and go that step size (unless you have learning rate), you usually can only converge to a local minimum

- We want to minimize the cost, so we can figure out which step to take (which new parameters to take on)

$C(w_1, \ldots, w_n)$ (C: cost)



- Similar for higher dimensions; for partial $\frac{\partial f}{\partial y}$ look at y-axis plane and take single variable derivative from there

$$\frac{\partial f}{\partial y} = \lim_{\Delta y \to 0} \frac{f(x, y+\Delta y) - f(x,y)}{\Delta y}$$

  - You are isolating the change to one variable at a time

- Backpropagation is the algorithm that computes the gradient

  - **Learning rate** is the step size

$$\theta_t = \theta_{t-1} - \eta \frac{\partial L}{\partial \theta_{t-1}}$$ (θ: weight matrix, η: dynamic, $\theta = \theta_{t-1}$, $\theta_t = \theta$) or $\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}$

  - Makes sense: $\frac{\partial L}{\partial \theta_{t-1}} = +$ ⇒ You want to decrease the parameter so you subtract it by this positive derivative

Also written as

$$W = W - \eta dW = w_{ij} - \eta \frac{\partial L}{\partial w_{ij}}$$ (matrix)

$$\vec{b} = \vec{b} - \eta d\vec{b} = b_i - \eta \frac{\partial L}{\partial b_i}$$

- You have a weight matrix and a bias vector for each pair of layers in a neural network

- Feed forward info → calculate cost → $C(\vec{w})$ is "starting point" → calculate $\frac{\partial C}{\partial \vec{w}} = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \vdots \end{bmatrix}$ (for x% of the training data; depends on type of gradient descent) → Update parameters which causes a "step" → step after certain number of iterations or convergence

$$-\nabla C(\vec{w}) = \begin{bmatrix} 2 \\ -2 \\ 0.21 \\ \vdots \\ 1 \end{bmatrix}$$ (all params)
- too much slope ⇒ decrease the parameter
- this weight change doesn't matter as much since that weight is close to a minimum

- Higher partial (just like normal derivative) means that the cost is very sensitive to that weight changing

- Doesn't seem like there's any pattern for changing the weights/biases

- **Batch Gradient Descent (BGD)** - take 1 "most accurate" step per epoch (control); converges slowly

- **Stochastic gradient descent (SGD)** - rather than computing loss function and gradient after using all training data you do it after each training example (lot of noise)

Mini-batch gradient descent

- Full batch gradient descent involves passing X (784×60000) through, calculating loss with Y (length 60000), calculating derivatives for each example and averaging, then taking one step

- For computational speedup, we can divide X and Y into $X^{\{i\}}_{mini}$ ($784 \times m$) and $Y^{\{i\}}_{mini}$ (length m), where m = batch_size
  - $\frac{num\_train\_samples}{m} = n$, the number of mini batches, where $i \in \{1, \ldots, n\}$
  - Per epoch, do * with mini batch matrices, then iterate to next mini batch matrices and repeat until the full batch of training examples have been used
    - More steps per epoch $\Rightarrow$ less epochs $\Rightarrow$ faster convergence $\Rightarrow$ SGD is more efficient

$\hat{\nabla L_B}(\theta_t) = \nabla L(\theta_t) + \varepsilon_t$ (estimate gradient of batch size B; ε_t: noise)

Variance of noise: $Var[\varepsilon_t] \propto \frac{1}{|B|}$ (inversely proportional)

If at local minimum... (local minimum $\Rightarrow \nabla L(\theta_t) = 0$)

$$\theta_{t+1} = \theta_t - \eta\left(\hat{\nabla L_B}(\theta_t)\right) = \theta_t - \eta\left(\nabla L(\theta_t) + \varepsilon_t\right) \qquad r = \theta_t - \eta(\varepsilon_t)$$

- For full batch size, $\theta_{t+1} = \theta_t$
- This is why the noise of smaller batch sizes can "kick" you out of local minima

- Theoretically, increase batch size $\Rightarrow$ more accurate avg of each partial $\Rightarrow$ more accurate trace gradient of loss function $\Rightarrow$ can take a bigger, more confident step $\Rightarrow$ safer to have higher learning rate
  - However, ↑ batch size $\Rightarrow$ less exploration of loss landscape

- Using .SGD in PyTorch with differing batch sizes determins the gradient descent used

- **Convex function** - only one global minima and no local minima
  - You ideally want convex loss function

- **Vanishing gradient problem** - Activation function has small derivative (ie. sigmoid) $\Rightarrow$ small numbers in chain rule being multiplied $\Rightarrow$ small $\frac{\partial C}{\partial \ell^{\{i\}}_{ij}} \Rightarrow$ small step size $\Rightarrow$ learns way too slow

- **Exploding gradient problem** - Activation function has small derivative (ie. ReLU) $\Rightarrow$ large numbers in chain rule being multiplied $\Rightarrow$ large $\frac{\partial C}{\partial \ell^{\{i\}}_{ij}} \Rightarrow$ large step size $\Rightarrow$ may skip minimum

Other Optimization Algorithms

Exponentially weighted moving average (EwmA)

- EwmA - "Exponentially B" results in prioritizing newer data points/averages; moving average because each average takes a step

$v_t = \beta v_{t-1} + (1-\beta)\theta_t$ calculates moving average of θ (v_t is arbitrary name for EwmA)

(v_{t-1}: EwmA at time $t-1$; θ_t: current value of a parameter)

- ↑B ↑ weight to recent observations; $0 \le \beta \le 1$ usually 0.9
- This applies to all parameters and can be stored in a vector $\vec{v}_{t+1}$

$v_0 = 0$

$v_1 = (1-\beta)\theta_1$ *

$v_2 = \beta v_1 + (1-\beta)\theta_1 = \beta(1-\beta)\theta_1 + (1-\beta)\theta_2 = (1-\beta)(\beta\theta_1 + \theta_2)$

$v_3 = \beta v_2 + (1-\beta)\theta_2 = (1-\beta)(\beta^2\theta_1 + \beta\theta_2 + \theta_3)$

(lower weight to old data point)

$v_6 = (1-\beta)\left(\sum_{i=0} \theta_i \beta^t\right)$

- * is too low if B is high (and since there is no average), can cross up future calculations
- Using bias correction, compute $\frac{v_t}{(1-\beta^t)}$ instead
  - low t ⇒ higher start (a boost when dividing)
  - higher t ⇒ $\lim_{t\to\infty} \frac{v_t}{(1-\beta^t)} = v^t$ ⇒ negligible effect

SGD with Momentum

- Smooths out the zig-zag path
- In regions with consistent gradients in contrast run of these "velocities" to get bigger and in same direction, leading to bigger step and faster convergence
- If loss landscape has gradients oscillating frequently (opposite signs), the previously averaged gradients result in taking a smaller step

$W = W - \eta v_{dW}$ (v_{dW}: EwmA of our weight)

$w := w - \eta v_{\partial L(\partial w)}$

$B = B - \eta v_{dB}$

t iterations

$v_{dw_t} = \beta * v_{dw_{t-1}} + (1-\beta)\,dW$ (v_{dW} means EwmA of dW, S_{dW} means $(dw)^2$; S stands for square)

$v_{dB_t} = \beta v_{dB_{t-1}} + (1-\beta)\,dB$

- EwmA will cancel out ∂_b and ∂_w (vertical motions) due to oscillations and converge faster
- No momentum for the momentum because that has unnecessary complexity and isn't really very beneficial

Root Mean Square Propagation (RMSprop)

- Without RMS bigger step taken in steeper places while smaller steps for smaller partials
- Uses an adaptive learning rate
  - For large gradients, "reduce" η to avoid overshooting
  - For small gradients, "increase" η to converge more quickly

- Involves squaring gradient components to focus on magnitudes
- Uses exponentially weighted moving average to prioritize more recent gradients

$$W = W - \eta \frac{dW}{\sqrt{S_{dw_t} + \varepsilon}} \quad \frac{\partial L}{\partial w} / \sqrt{S_{dw}}$$

$$S_{dw_t} = \beta S_{dw_{t-1}} + (1-\beta)(dW)^2 \quad \nabla L(w)$$

(S_{dw_t}: EWMA of dw^2; β: decay rate)

$$B = B - \eta \frac{dB}{\sqrt{S_{dB} + \varepsilon}} \quad \varepsilon \text{ usually } 10^{-8}$$

(adds to small denominator so you don't take big step or overshoot)

$$S_{dB_t} = \beta S_{dB_{t-1}} + (1-\beta)(dB)^2$$

(β: 0.999 usually)

or

$$W = W - \frac{\eta}{\sqrt{S_{dw_t} + \varepsilon}} \nabla L(w)$$

(η: adaptive η)

$$S_{dw_t} = \beta S_{dw_{t-1}} + (1-\beta)(\nabla L(w))^2$$

- $\uparrow dW \Rightarrow \uparrow S_{dw} \Rightarrow \downarrow \frac{dw}{\sqrt{S_{dw}}}$
- Steep direction $\Rightarrow$ large $\frac{\partial L}{\partial w_{ij}} \Rightarrow$ high moving average $\Rightarrow$ smaller step in this axis
- RMSprop focuses on gradient magnitude/step size while momentum focuses on the sign/direction of step

Adam (Adaptive Moment Estimation)

$$W = W - \frac{\eta}{\underbrace{\sqrt{S_{dw} + \varepsilon}}_{\text{RMS}}} \underbrace{V_{dw}}_{\text{Momentum}} \quad \text{or } \nabla L(w)$$

$$B = B - \frac{\eta}{\sqrt{S_{dB} + \varepsilon}} V_{dB}$$

$$V_{dw_t} = \beta_1 V_{dw_{t-1}} + (1-\beta_1) dW$$

$$S_{dw_t} = \beta_2 S_{dw_{t-1}} + (1-\beta_2)(dW)^2 \quad \nabla L(w)$$

(S_{dw_t}: EWMA of dw^2; β_2: decay rate)

- β_1 is the momentum parameter, β_2 is the RMSprop parameter
- V_{dw_t} called m_t, the first moment, and S_{dw_t} is second moment
  - Moment captures characteristic of gradient distribution overtime

Both of these use bias correction so they don't favor zero

- $m_{t_{corr}} = \frac{m_t}{1-\beta_1^t}$

- Adam usually better than SGD with momentum or RMS prop alone

- Autograd engine computes tensor gradients; it traces through computation graph

Learning Rate Optimizer/Scheduler

- Optimization algorithms like Adam work to adjust learning rate differently for each parameter (local adjustments)
  - Adam uses momentum for the momentum and RMSprop to adjust the step size taken in direction of each parameter

- There are also changes that should apply globally to the step size at certain points in the training process
  - **Learning rate schedulers** will change the learning rate efficiently every n epochs to converge faster and not get stuck in local minima/saddle points
    - Can complement optimization algorithms well

- **Step decay** - learning rate reduced by fixed amount (gamma γ) every n epochs (step size)
  - Makes sure you don't overshoot and take smaller steps for all parameters as they get closer to minimum

- $\gamma = 0.1 \Rightarrow$ learning rate is 10% of its previous value

$\lfloor x \rfloor$ = floor(x)

$$\eta_n = \eta_0 \times \gamma^{\left\lfloor \frac{n}{\text{step\_size}} \right\rfloor}$$

(n = current #)

Say $\eta_0 = 1$ and step_size = 10

Epoch 0-9 (10 epochs): $\eta_9 = 1 \cdot (0.1)^{\left\lfloor \frac{9}{10} \right\rfloor} = 1 \cdot (0.1)^0 = 1$ (10% of previous value)

Next 10 epochs: $\eta_{11} = 1 \cdot (0.1)^{\left\lfloor \frac{11}{10} \right\rfloor} = 0.1$

- **Exponential decay** decreases exponentially over epochs

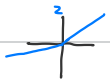
Functions

ReLU(z) $= \max(0,z) = \begin{cases} 0 & z \le 0 \\ z & z > 0 \end{cases}$ $\quad relu'(z) = \begin{cases} 0 & z \le 0 \\ 1 & z > 0 \end{cases}$

- Most people use ReLU instead of sigmoid now because it is simple and $\sigma'(z) = \{1$, which mitigates vanishing gradient problem
  - **Dying ReLU problem**: many negative $z \Rightarrow \sigma'(z) = 0 \Rightarrow$ impacts portion of previous layers' parameters $\Rightarrow$ stops learning / slows

To solve this, leaky $\text{RELU}(z) = \max(\alpha z, z) = \begin{cases} \alpha z & z \le 0 \\ z & z > 0 \end{cases}$ $\quad \sigma' z = \begin{cases} \text{small} \\ 1 \end{cases}$

(α: small positive like 0.01)



  - Small $\alpha \Rightarrow$ small $\sigma'(z)$ & neurons don't die

Cross-entropy loss $(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log_e(\hat{y}_i)$ (C: # of classes / output neurons; y_i: 0 or 1; $\ln$)

only term left is $-\log(\hat{y}_i)$ of the correct classification

- Cross-entropy good for classification because it focuses on correct class and wrong classification $\Rightarrow$ high loss; punishes this (squared penalizes this)
- $-$ because $\log_e(x)$ when $x < e$ is negative so you have to make the loss non-negative function to minimize it with gd
- e and $\ln$ used because they simplify derivatives for chain rule
- For whole batch take average of the losses

Mean Squared Error (MSE) $= \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

- For mini-batch GD, it is common to track average loss over each epoch instead of final loss; epoch loss defined as average batch loss in the epoch
  - This can stabilize the loss resulting from step of outlier batch; avg-loss per epoch $= \frac{\text{total epoch loss} \leftarrow \sum \text{batch}_i \text{ loss}}{\text{num batches}}$
  - Even though average loss per epoch doesn't go down as fast as loss at end of epoch, it can be more consistent, smooth, and reliable results can be generalized on whole training data not just last batches of epoch

Softmax activation function

- For output neurons before computing loss they're passed through **softmax function** that maps results between 0 and 1 (probabilities)
  - The raw "scores" passed to softmax function are called **logits** (not related to stats)

probability of this classification from output layer $= \sigma(\vec{z})_i = \frac{e^{z_i/T}}{\sum_{j=1}^{C} e^{z_j/T}}$ (i: pick of ith output neuron; C: # of classes; T: optional temperature value)

- Ensures the outputs add to 1, larger values are closer to 1; $(\text{positive number})^x > 0$, ensures all the probabilities are positive

- **Sigmoid function**, $\sigma(x) = \frac{1}{1+e^{-x}}$ does this more efficiently for binary classification
  - Sigmoid does not rely on other neuron outputs from the output layer
  - E.g. $\vec{z} = \begin{bmatrix} z_0 \\ z_1 \end{bmatrix}$

    $\text{Softmax}\left(\begin{bmatrix} z_0 \\ z_1 \end{bmatrix}\right) = \begin{bmatrix} p_0 \\ p_1 \end{bmatrix}$ (p_1: prob of positive class)

    $\text{Sigmoid}(z_1) = p_1$

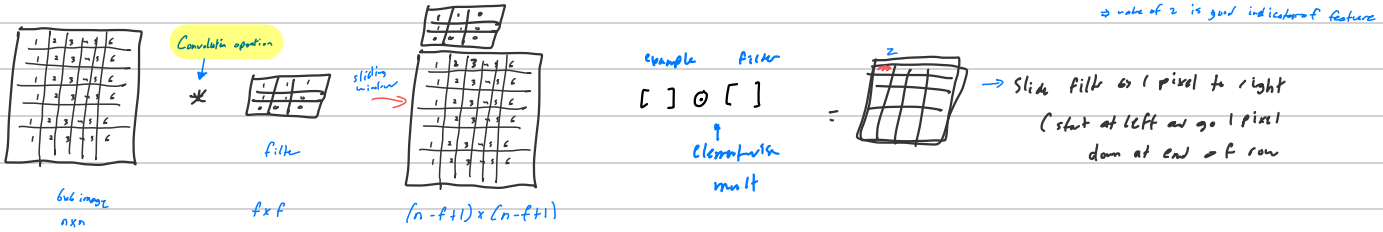
Convolutional Neural Network (CNN)

- Useful for image recognition
- Large images ⇒ many features ⇒ too many parameters ⇒ computationally expensive and overfitting

- **Filter/convolution kernel** - sliding window consisting of certain # of pixels for detecting a feature/pattern
  - Each  is a parameter

Low loss ⇒ good w/ input ⇒ each z is good ⇒ good weights in filter

⇒ value of z is good indicator of feature



- The **convolution** operation produces one pixel of output feature map at a time
  - Convolutional layer directly takes in 2D image as input, no need to flatten

- Different filters detect different things (discovered by mathematicians);
  - Eg edge detection; transpose of horizontal filter is the vertical filter

- Many filters in each layer of the neural network that each produce an image

$(n \times n \times 1) * (f \times f \times c) \rightarrow (n-f+1) \times (n-f+1) \times c$ when stride = 1

(n x n x 1): grayscale; f x f: filter size; c: number of filters; c: number of images produced - 1 per filter

- Colored image is RGB, $(n \times n \times 3)$ size ⇒ filter is $f \times f \times 3$
  - Same sliding window concept applies except the sliding window is a cube (to output one feature map element)
  - Applies to higher dimensions

- Each element of filter is a parameter that is trained

- Each convolutional layer passes outputs/feature maps of previous layer
  - Each layer can detect different/more complex features

- After convolution, bias is added to new image and you pass through activation function

Padding

- Each convolution reduces image size
  - If you have too many layers where the output is passed to another filter, it may get too small and lose info
- Certain pixels (like corner) only passed through filter once

- **Padding** - surrounding image with certain width of zeros

- **Valid convolution** - no padding
- **Same convolution** - image $\xrightarrow{\text{convolution}}$ feature map of same size since $n' - f + 1 = n$ (n' = new image size after)

$n' = n + 2p$ (p = padding)

$n' - f + 1 = n$

$n + 2p - f + 1 = n$

$p = \frac{(f-1)}{2}$ must be true for same convolution $\Rightarrow$ for p to be whole number f must be odd $\Rightarrow$ f usually odd for same convolution

Strides

- **Stride** - the amount of pixels the filter moves right/down
- Convolution called **strided convolution**

- This could discard all the pixels at the bottom
  - Size of final image = $\text{floor}\left[\frac{n - f}{s} + 1\right] \times \text{floor}\left[\frac{n+p-f}{s} + 1\right]$ (n = input size, f = filter size, s = stride)
- If you double the stride it halves the dimension of output feature map

Pooling Layer

- Purpose of **pooling layer** is to reduce dimensions of image while preserving features
  - Smaller image $\Rightarrow$ less computational cost
  - Enhances/sharpens features

Image $\rightarrow$ convolutional layer $\rightarrow$ pooling layer (usually after 1 or more convolutional layers)

- For nxn pooling matrix, n = stride so pooling windows don't overlap $\Rightarrow$ no redundancy in extracted features
  - **Max pooling** - takes max value in the sliding window
  - **Average pooling** - takes average of values in sliding window
  - Doubling stride halves n of resulting feature map

- No parameters involved in pooling (no training required)

Multilayer Perceptron

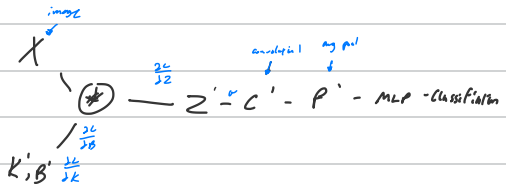
- The end of a CNN has a fully connected (FC) multilayer perceptron
- Fully connected layer - every neuron in layer ℓ connects to every neuron in layer $\ell \pm 1$
- All transformed feature maps from convolving/pooling flattened into a vector → passed as input to NN
- This associates the extracted features to a label
- Conv + Pooling = 1 layer
- Backprop very similar except now you have filter parameters

Backprop

X = Image

K = Filter/Kernel

B = Bias



- Since the filter in one section of image is one element of Z, you can think of that element of Z, once CNN trained, as a "feature" detector for that part of the image
- Adjusting the weights in the filter so Z can be a feature detector

Confusion Matrix and Model Analysis

- Element c_{ij} of confusion matrix C is equal to the number of examples labeled as class i predicted to be in class j

- **Macro averaging** - involves calculating metric for each class independently then taking average; $\frac{1}{\#\text{ classes}} \sum \text{stat}_i$
  - Better for treating classes equally regardless of size

- **Micro averaging** - aggregate/sum up results from all classes at once to get metric representative of entire model; better for imbalanced classes
  - Bigger categories have bigger impact; **weighted averaging** accounts for this for imbalanced classes also

$$\sum \frac{x_i}{x_i + y_i} \neq \frac{\sum x_i}{\sum x_i + y_i} \qquad \text{If formula} = \frac{x}{x+y}\text{, micro averaging is } \frac{\sum x_i}{\sum x_i + y_i}$$

- **Support** - # of instances per class; used in weighted averaging

- Multiclass score calculations applied per label and then some sort of average is taken

$TP + FN$ = All examples in positive class

$TN + FP$ = All examples in negative class

- **Accuracy** - measures overall class of the model
  - Binary

$$\text{Accuracy} = \frac{\text{correct classification of positive and negative class}}{\text{total}} = \frac{TP + TN}{TP + TN + FP + FN}$$

  - Multiclass

$$\text{Accuracy} = \frac{\sum \#\text{correct classifications for class } i}{\sum \text{total classifications for class } i}$$

- **Precision / Positive Predictive Value** - measures accuracy of the positive predictions (focuses on just positive class, unlike accuracy) "how many animals that you classified as 2 are actually a 2"
  - Binary

$$\text{Precision} = \frac{TP}{TP + FP}$$

Precision = 1 ⇒ every example that the model classified as positive class was correct, but it may have misclassified other examples of that positive class (FN)

  - Multiclass
    - **Macro Precision** - Calculate precision for each class then take the average

$$\text{Precision}_{macro} = \frac{1}{\#\text{ classes}} \sum_{i=1}^{n} \text{Precision}_i$$

    - **Micro Precision** - Aggregate the contributions of all classes to compute average precision

$$\text{Precision}_{micro} = \frac{\sum TP_i}{\sum (TP_i + FN_i)}$$

- **Recall / True Positive Rate / Sensitivity** - measures ability of the model to correctly identify actual positives; out of all examples in positive class, what proportion is correctly identified.

TP + FN = Total examples labeled as positive class

- Binary

$$\text{Recall} = \frac{TP}{TP+FN}$$

(TP: total identified by model as positive correctly)

- Multiclass
  - Micro and macro recall
  - Micro Precision = Micro Recall due to a FP in 1 class is FN in another class

- **False Negative / Miss Rate** - complement of recall; out of all examples in positive class, what proportion is missed / incorrectly identified

$$FNR = \frac{\text{Incorrectly identified of class}}{\text{Total \# of class}} = \frac{FN}{TP+FN}$$

$$FNR + TPR = 1$$

- **F1 Score** - harmonic mean of precision and recall; provides a balance between the two
  - **Harmonic mean** = $\frac{2ab}{a+b}$; gives more weight to smaller numbers; if a or b is small, the harmonic mean is also small

$$F_1 \text{ Score} = 2\frac{(\text{Precision})(\text{Recall})}{\text{Precision} + \text{Recall}}$$

- In general F (formula) - measures evaluate performance of classification models by combining precision and recall with different weights; score parametrized by β
  - $F_\beta = (1+\beta)^2 \cdot \frac{\text{Precision} \times \text{Recall}}{\beta^2 \text{Precision} + \text{Recall}}$; $\beta = 1 \Rightarrow$ Precision and recall given equal weight; $\beta > 1 \Rightarrow$ more weight to precision

(1 is best, 0 is worst)

- Multiclass
  - Macro F1 Score and Micro F1 Score

- **True Negative Rate / Specificity** - out of all the examples in negative class, what proportion is correctly identified

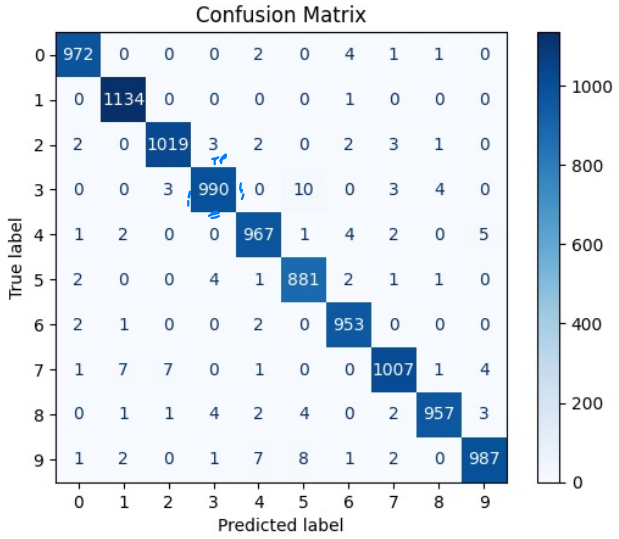
$$TNR = \frac{TN}{TN+FP}$$

- **False Positive Rate** - complement of TNR; out of all the negative class, how many are falsely identified

$$FPR = \frac{FP}{TN+FP}$$

$$TNR + FPR = 1$$

Example Calculations



- Per class, look at the column for accuracy and row for recall

Class 3

10,000 = # total examples in dataset = 3's + non 3's = (TP + FN) + (TN + FP)

$TP = 990$ — "how many actual 3's did model correctly identify/classify"

↑ c_{ii} in diagonal

$FN = 3+10+3+4 = 20$ — "how many 3's did the model fail to identify"

↑ the row

$TP+FN = 1010$ — "how many 3's were in the dataset"

$FP = 1+4+4+3 = 12$ — "how many non 3's were incorrectly identified as a 3"

↑ column

$TN = \text{Total} - (TP+FN+FP) = 8,978$ — "how many non 3's were correctly identified as non 3" ← doesn't mean they were classified as correct number

$FP+TN = 8,990$ — "how many non 3's are in the dataset"

~

$$\text{Accuracy}_3 = \frac{\text{Correct \# of identifications of 3's and not 3's}}{\text{total \# of 3's and not 3's}} = \frac{TP+TN}{TP+TN+FP+FN} = \frac{990+8,978}{10,000} = \frac{9,968}{10,000} = 0.9968$$

- Model good at making classification of 3 or not 3

$$\text{Precision}_3 = \frac{\text{Correct \# of identifications of 3's}}{\text{Total identification of 3s}} = \frac{TP}{TP+FP} = \frac{990}{990+12} = 0.9880$$

- Of the examples the model classifies as 3, 98.8% are actually a 3 and 1.12% of those are false positives

- Model over predicting class 3 ⇒ ↑ FP ⇒ ↓ Precision

$$\text{TPR: Recall}_3 = \frac{\text{Correctly identified 3s}}{\text{Total 3s}} = \frac{TP}{TP+FN} = \frac{990}{1010} = .9802$$

- Out of all the threes, the model correctly classifies 98.02% of them

FNR = .0198 Out of all 3s being classified by model 1.98% are incorrectly classified as not 3

- False Positive Rate - out of all the not threes, how many does the model falsely classify as a 3

$$FPR = \frac{FP}{TN+FP} = \frac{12}{8990} = 0.001$$

TNR = 0.999 model really misclassifies other numbers as 3

- We prioritize low number of false positives (e.g. spam, want important stuff not to spam) ⇒ low $FPR = \frac{FP}{TN+FP}$ ⇒ precision dominator decreases ⇒ higher precision
- we prioritize low number of false negatives (e.g. medicine) ⇒ low $FNR = \frac{FN}{TP+FN}$ ⇒ high recall (numerator)

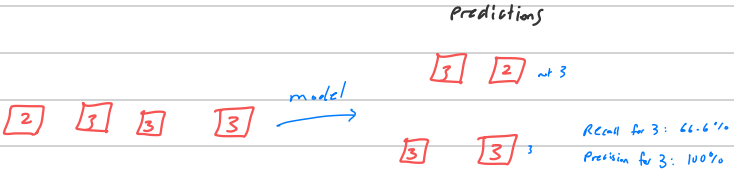
- For something like medicine you prioritize lower # of false negatives; for number classification it's not as important

$$F_1 \text{ Score} = \frac{2(\text{Precision})(\text{Recall})}{\text{Precision} + \text{Recall}} = 0.984$$

model has good balance of correctly identifying the number 3 (recall) and making sure its predictions are accurate (precision)

Ex.

Data:



- In addition to each class analysis, aggregate results for whole model performance with macro, micro, or weighted averaging

Colab/Metal

- **Server** - Computer that provides resources, data, and services to other computers (**clients**) over a network (collection of connected devices)
  - If server breaks you are screwed

- **Cloud**
  - When you store something it is saved in various servers in **data centers** (controlled/managed etc.) so there are backups
  - When you "request resources" to do something from cloud infrastructure, you are given part of a CPU
    - Hypervisor allocates certain amount of physical resources like CPU per person running

- **TPU (Tensor Processor Unit)** - a hardware accelerator unit (does something specific very fast)
  - Open Colab → one HTTP request sent to servers and around I get access → request to allow for TPU sent & allocate resources → I get results back
  - TPU fast for parallelism, matrix multiplication
    - Since it's very parallel you should mark end of calculation step (**mark_step()**); could otherwise lead to **race conditions** (operation starts before previous one finishes) and poor memory management
    - How to warm up TPU so it establishes data paths so it runs quickly

- **xm.xla_device()** sets up communication between CPU and TPU
- **.to(device)** moves tensor from virtual machine memory (mapped to RAM of server) to TPU memory (or whatever backend)

CPU memory: registers, caches, RAM

TPU memory: Scratchpad/SRAM, unified buffers, high bandwidth memory
- ↑ for reused data like weights/activations
- ↑ good for large models/datasets

- **Metal API** - software that directly interacts with the hardware (GPU) (contains the functions/classes to utilize Metal framework)
  - **Backend** - system that performs the actual computation (operations on tensors)
  - You choose a device/backend to execute your code on
  - This process uses the **Metal API**, which schedules how to send/store the GPU operations
    - **Command Queue** holds **Command Buffers** (sequences of commands)
  - **torch.device("mps")** Creates MTLDevice in Metal and creates these metal objects to transfer tensor
    - Pytorch encodes commands for computation with Metal **Compute pipeline**

- Compute Pipeline
  - **Compute shader/kernel** - a program written in Metal Shading Language (MSL) for data-parallel GPU computations
    - **Metal Performance Shaders (MPS)** (part of API) include pre-optimized kernels for common ML operations like matrix multiplication
  - **Pipeline state object (PSO)** properly configures shader
  - Command buffers/queues with resources

- GPU's and TPU's benefit from large batch size since all examples in batch size passed through NN at once (parallelism)
  - Sometimes overhead of transfering data to GPU/TPU doesn't pay off (can mitigate by sending next batch to memory while calculating current one)
  - Simple dataset (ie. binary classes) ⇒ simple model with few layers/neurons ⇒ not many computations to be done in parallel (small weight matrices/biases makes; calculating less components in parallel) ⇒ not much benefit from GPU/TPU

  - Small dataset ⇒ small batch size (to add noise to steps (form of regularization) to prevent overfitting) ⇒ less parallelism ⇒ less benefit from GPU/TPU
    - ↓ more frequent steps + faster convergence

Multiprocessing

- When getting an item from dataset, all transformations/preprocessing is applied
  - DataLoader loads batch from dataset → for each example the dataset's `__getitem__` method is called → transformations applied → data points batched together → data moved to GPU
    - ? How many workers → more CPUs load at once
    - more of next ready with prefetch
    - does this at same time if num_workers > 0
    - worker does up to here
- The goal is to make sure the GPU is being fed data and isn't idle while CPU preprocesses the data

- **Page-locked/pinned memory** - region of memory that OS guarantees will not be swapped out to disk
  - sometimes it does this if RAM space is low
  - Speeds up transferring data from CPU to GPU if tensors put here
  - Allows CPU to load/preprocess next batch of data while current batch being processed by GPU; good for asynchronization otherwise GPU is waiting
    - Metal makes sure this asynchronization is handled properly and CPU doesn't give GPU new data until it's done

- **Non-blocking = True** means CPU & GPU data transfer is **asynchronous**
  - CPU can do other stuff
  - CPU will preprocess data of new batch while the previous batch is being transferred to GPU; CPU could technically return function values or move on before GPU is done calculating so you should synchronize
- 0 workers ⇒ data loading/preprocessing occurs in **main process** ⇒ the CPU core that runs the main script preprocesses the data
  - No inter-process communication
  - The main process always handles transfer to GPU

- > 0 workers ⇒ workers independently process data and utilize CPU cores ⇒ combine batch into a queue shared with the main process
  - `Persistent_workers=True` keeps workers throughout all epochs rather than remaking them
  - Each worker process initialized with copy of dataloader and works independently
  - Queue allows for **inter-process communication (IPC)**

- **pre-fetch factor** - the amount of batches each worker process preloads for the main process to get
  - Workers need to serialize and deserialize dataset to create separate worker processes
  - Putting data in separate script makes it easier for worker processes to find the class definition
  - Warning w DataLoader creates workers and gives them the initial data to prepare and each move data into memory; communication channels also has to be set up

- **Global Interpreter Lock** (GIL) prevents true parallelism because Python memory management isn't thread-safe; instead uses **concurrency**
  - tracking turns
  - Can also bypass by giving each process its own Python interpreter/GIL; `multiprocessing` module does this
  - Slightly different than **multithreading** since its not one process with multiple threads sharing memory

Variations in Results

- Floating point operations (especially lots of them in neural network) can introduce tiny error in rounding over many iterations which can result in slight differences
- Parallel computing can sometimes affect results since some floating point arithmetic is not associative (i.e. $(a+b)+c \neq a+(b+c)$ sometimes due to rounding)
  - when rounding
- Even with setting seeds this error can still occur
  - Same data and code $\not\Rightarrow$ same errors

Compilers

- JIT compiler translates high-level code into machine code, and it can replace function call with whole function
  - First time you JIT-compile a function, it "traces" the initial parameters and sets up and caches it
    - In PyTorch, TorchScript has a JIT compiler
  - If you call function again with different parameters of same type/shape, it can use the cached machine code
  - Usually you just turn model to Torchscript model; more optimization soon

- Torchscript provides intermediate representation of model and can optimize computation graph, which is good for GPU
  - Graph base representation of model means pytorch can fuse different operations into 1 kernel (more kernels ⇒ overhead to launch them on GPU)
    - Python computation graph made dynamically (not at once) so how to tell what kernels can be combined
  - Removes parts of computation graph that don't effect output (dead code)
  - Reuses memory allocations

- Python interpreter reads and executes one line at a time; the bytecode doesn't map to CPU registers
  - source code → bytecode → interpreter reads → machine code → executes

- Module is base class for all neural network layers and models contained in torch.nn module; classes that extend this are modules
  - Can be a single layer or container of them (Sequential)
  - Module contains parameters of model and methods to initialize them; also implements forward pass of network
  - Any nested Module in a larger module is called a submodule