# ECE C147 Cheat Sheet

## Linear Algebra

1. $tr(A) = tr(A^T), tr(ABC) = tr(CAB) = tr(BCA)$
2. $tr(aX + bY) = atr(X) + btr(Y)$. $tr(Scalar) = Scalar$.
3. $x^T y = ||x||_2 ||y||_2 cos(\theta)$, $||x||_2 = \sqrt{x^T x}$
$||x||_p = (\sum_{i=1}^n |x_i|^p)^{\frac{1}{p}}$
4. Eigendecomposition (when the rank is $n$ for a $n \times n$ matrix): $A = U \Lambda U^{-1}$ where $U$ is a matrix in which the columns are the eigenvectors of $A$, and the eigenvalues of $A$ are in the diagonal of the diagonal matrix $\Lambda$. If $A$ is normal ($AA^T = I$) then its eigenvectors are orthonormal.
5. Singular value decomposition (can always be performed on $A \in \mathbb{R}^{m \times n}$): $A = U \Sigma V^T$ where $U \in \mathbb{R}^{m \times m}$ is with orthonormal columns (the left singular vectors of $A$), $V \in \mathbb{R}^{n \times n}$ is with orthonormal columns (the right singular vectors of $A$), and $\Sigma \in \mathbb{R}^{m \times n}$ is a diaginal matrix with the singular values $\sigma_i$ of $A$ in the diagonal.
6. The Frobenius norm of a matrix $A \in \mathbb{R}^{m \times n}$ is: $||A||_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2} = \sqrt{tr(AA^T)}$.
7. A matrix $A$ is called positive definite if $x^T \boldsymbol{A} x > 0$ for all $x$. In this case all of the eigenvalues are positive. If $x^T Ax \geq 0$ for all $x$, then $A$ is positive semidefinite, and all of the eigenvalues are not negative.

## Probability Theory

1. Notation: $Pr(X = x) = p(x)$
2. $p(x) = \sum_y p(x, y)$, $x$ and $y$ are discrete. $p(x) = \int_y p(x, y) dy$, $x$ and $y$ are continuous.
3. Chain rule: $p(x, y) = p(x)p(y|x) = p(y)p(y|x)$. More general: $p(w, x, y, z) = p(w, x)p(y, z|w, x)$.
4. Bayes' rule: $p(x|y) = \frac{p(y|x)p(x)}{p(y)}$
5. Likelihood - $P(D|\theta)$ - is the probability to observe our data given the parameters we chose for our model.

## Multivariate Calculus

1. The gradient of a scalar $y$ with respect to $\boldsymbol{x}$ has the same dimensionality as $x$ (even if $\boldsymbol{x}$ is a scalar, vector, or a matrix).
2. $\nabla_x \theta^T x = \theta$
3. $\nabla_x x^T \boldsymbol{A} x = (\boldsymbol{A} + \boldsymbol{A}^T) x$

## Multivariate Calculus

4. If $y \in \mathbb{R}^n$ and $x \in \mathbb{R}^m$, then $(\nabla_x y)^T \in \mathbb{R}^{n \times m}$:
$J = \begin{pmatrix} (\nabla_x y_1)^T \\ \vdots \\ (\nabla_x y_n)^T \end{pmatrix}$. This is the Jacobian matrix. Notice that: $J = (\nabla_x y)^T$
5. $\nabla_x W x = W^T$ where $W$ is a matrix.
6. The Hessian matrix of a function $f(x)$ is a square matrix of second-order partial derivatives of $f(x)$. $H = \nabla_x (\nabla_x f(x)) = \nabla_x^2 f(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1^2} & \cdots & \frac{\partial f(x)}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(x)}{\partial x_n \partial x_1} & \cdots & \frac{\partial f(x)}{\partial x_n^2} \end{pmatrix}$
7. Let $x \in \mathbb{R}^m, y \in \mathbb{R}^n, z \in \mathbb{R}^p$, and $y = f(x), z = g(y)$. Thus, $\nabla_x z = \nabla_x y \nabla_y z$. In the "denominator" layout, the chain rule runs from right to left. $\nabla_x z$ have dimensionality $\mathbb{R}^{m \times p}$.
8. $\nabla_x \boldsymbol{W} x = \boldsymbol{W}^T$
9. In the denominator layout, the chain rule runs from right to left: $\nabla_x z = \nabla_x y \nabla_y z$
10. $\frac{\boldsymbol{\partial z}}{\boldsymbol{\partial x_i}} = \sum_{\boldsymbol{j}} \frac{\boldsymbol{\partial y_j}}{\boldsymbol{\partial x_i}} \frac{\boldsymbol{\partial z}}{\boldsymbol{y_j}}$
11. $\nabla_x x^T x = 2x$

## Overfitting and underfitting

The problem is that more complex models may not generalize as well if the data come from a different model. **Training data** is data that is used to learn the parameters of your model. **Testing data** is data that is used to score your model. **Validation data** is data that is used to optimize the hyperparameters of your model. This avoids potential overfitting to nuances in the testing dataset.
A model with low training error but high testing error is overfitting. A model with high training error and high testing error is underfitting. Large datasets allow the use of more complex models, because then they are less sensitive to overfitting. **Model selection** is a way to to penalize models for being overly complex.

## Regression (least-squares)

We define the cost function:

$$\mathcal{L}(\theta) = \frac{1}{2} \sum_{i=1}^N (y^{(i)} - \theta^T \hat{x}^{(i)})^2$$

$\theta = (X^T X)^{-1} X^T Y$ minimizes $\mathcal{L}(\theta)$ where $X = \begin{pmatrix} (\hat{x}^{(1)})^T \\ \vdots \\ (\hat{x}^{(N)})^T \end{pmatrix}$ and $Y = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{pmatrix}$. This works for any degree polynomial of $x$.

## k-fold cross validation

Suppose the training dataset have $N$ examples, and that we have a separate testing dataset. Split the training dataset into $k$ equal sets, each of $N/k$ examples. Each of these sets is called a "fold". $k-1$ of the folds are datasets that are used to train the model's parameters. The remaining fold is a validation dataset. If the model have hyperparameters, the remaining fold helps us choose the best hyperparameters so that they don't overfit to the actual test dataset.

## Maximum Likelihood Estimation

Suppose each point in our dataset is independent. The likelihood is the probability to observe our data:

$$\mathcal{L} = Pr(D = d) = p(\{x^{(1)}, y^{(1)}\}, ..., \{x^{(N)}, y^{(N)}\})$$

$$= \Pi_{i=1}^N p(x^{(i)}, y^{(i)}) = \Pi_{i=1}^N p(x^{(i)}|y^{(i)}) p(y^{(i)})$$

. To maximize the likelihood, we can maximize the log-likelihood since log is a monotonic function:

$$\log \mathcal{L} = \sum_{i=1}^N [\log p(y^{(i)}) + \log p(x^{(i)}|y^{(i)})]$$

If we assume that each label $y^{(i)}$ follows some distribution (for example $x^{(i)}|[y^{(i)} = 1] \sim \mathcal{N}(\mu_1, \Sigma_1)$), then after maximizing the likelihood and given a new data point $x$, we can determine the probability that $x$ is in class 1: $p(y_x = 1|x)$ by using Bayes' rule.

## Supervised classification

Several challenges associated with image classification: Viewpoint variation, Illumination, Deformation (for example the cat is flexing his body), Occlusion (for example, we can't see the whole cat), Background clutter (for example, in one image the cat is on a tree and on the other the cat is in a house), Intraclass variation (many kinds of cats). In this class, we use data driven approach to classify signals (such as images).

**K-Nearest Neighbors**

Find the $k$ closest points to $x_{new}$ - the signal you wish to classify - in the training set, according to an appropriate metric. Each of its $k$ nearest neighbors then vote according to what class it is in, and $x_{new}$ is assigned to be in the class with the most votes.

- Choose an appropriate metric could be $d(x^{(i)}, x^{(j)}) = ||x^{(i)} - x^{(j)}||$, for example.
- Choose the number of nearest neighbors, k.
- Take the desired test data point, $x_{new}$, and calculate $d(x_{new}, x^{(i)})$ for all $i = 1, \ldots, m$.
- With $(c_1, \ldots, c_k)$ denoting the $k$ indices corresponding to the $k$ smallest distance, classify $x_{new}$ as the class that occurs most frequently among $(y^{c_1}, \ldots, y^{c_k})$. If there's a tie, any of the tying classes may be selected.

**Pros**: Simple to test, simple to train. Training complexity is $O(1)$. **Cons**: Test complexity is $O(NlogN)$ where $N$ is the number of data points.
In general, we prefer better test complexity, and it's okay if training takes long. The **hyperparameters** of KNN are $k$ and the distance metric.
**KNN might not be a good idea for image classification** because of "the curse of dimensionality". As our feature vectors increase in dimensionality, the "volume"that our vectors occupy grows exponentially. There's a lot of "empty space"in high dimensional data. Thus, our notion of distance begins to break down. Another problem with KNN is that shifting an image for example would result in a significance difference between images.

**Softmax classifier**

The softmax function transforms the class score to a probability: $\boxed{softmax_i(x) = \frac{e^{a_i(x)}}{\sum_{j=1}^{c} e^{a_j(x)}}, a_i(x) = w_i^T x + b_i}$ where $c = Num\ of\ classes$. $softmax_i(x)$ gives the probability that $x$ is in class $i$: $Pr(y^{(j)} = i|x^{(j)}, \theta) = softmax_i(x^{(j)})$. We want to choose $\theta$ so as to maximize the likelihood of having seen the data. Assuming the samples $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$ are $i.i.d$, this corresponds to maximizing: $p(x^{(1)}, \ldots, x^{(m)}, y^{(1)}, \ldots, y^{(m)}|\theta) = \Pi_{i=1}^m p(x^{(i)}, y^{(i)}|\theta) = \Pi_{i=1}^m p(x^{(i)}|\theta)p(y^{(i)}|x^{(i)}, \theta)$. Therefore: $\arg\max_\theta \Pi_{i=1}^m p(x^{(i)}|\theta)p(y^{(i)}|x^{(i)}, \theta) = \arg\max_\theta \Pi_{i=1}^m p(y^{(i)}|x^{(i)}, \theta) = \boxed{\arg\min_\theta \sum_{i=1}^{m} [\log \sum_{j=1}^{c} e^{a_j(x^{(i)})} - a_{y^{(i)}}(x^{(i)})]}$.

**Overflow of softmax**: A potential problem when implementing a softmax classifier is overflow. If $a_i(x) \gg 0$, then $e^{a_i(x)}$ may be very large, and numerically overflow. Thus, it's standard practice to normalize the softmax function as follows: $softmax_i(x) = e^{a_i(x)+\log k} / \sum_{j=1}^{c} e^{a_j(x)+\log k}, k = -max_i a_i(x)$, making the maximal argument of the exponent to be 0.

**Support vector machine**

Informally, the SVM finds a boundary that maximizes the margin (gap) between the boundary and the data points. If a point is further away from the decision boundary, there ought to be greater confidence in classifying that point. **Hinge loss**: the hinge loss function is standardly defined for a binary output $y^{(i)} \in -1, 1$. If $y^{(i)} = 1$, then we want $w^T x^{(i)} + b$ to be large and positive, and if $y^{(i)} = -1$, then we want $w^T x^{(i)} + b$ to be large and negative:

$$hinge_{y^{(i)}}(x^{(i)}) = max(0, 1 - y^{(i)}[w^T x^{(i)} + b])$$

## Supervised classification

There are a few things to notice about the form of this function:

- If $w^T x^{(i)} + b$ and $y^{(i)}$ have the same sign, indicating a correct classification, then $0 \leq hinge_{y^{(i)}}(x^{(i)}) \leq 1$. The error will be 0 if the score is large (corresponding to a large margin), and will be nonzero if the score is small.
- If $w^T x^{(i)} + b$ and $y^{(i)}$ have the opposite sign, then $hinge_{y^{(i)}}(x^{(i)}) \geq 1 + |w^T x^{(i)} + b|$.

**Extension of SVM to multiple classes**: An extension of the hinge loss to multiple potential classes is the following loss:

$$hinge_{y^{(i)}}(x^{(i)}) = \sum_{j \neq y^{(i)}} max[0, 1 + a_j(x^{(i)}) - a_{y^{(i)}}(x^{(i)})]$$

Where $a_j(x^{(i)}) = w_j^T x^{(i)}$. Some intuitions, for the scenario where there are $c$ classes:

- When the correct class achieves the highest score, $a_{y^{(i)}}(x^{(i)}) \geq a_j(x^{(i)})$ for all $j \neq y^{(j)}$, then $a_j(x^{(i)}) - a_{y^{(i)}}(x^{(i)}) \leq 0$ and $\boxed{0 \leq hinge_{y^{(i)}}(x^{(i)}) \leq c - 1}$.
- When an incorrect class $i$ achieves the highest score, then $a_j(x^{(i)}) - a_{y^{(i)}}(x^{(i)}) \geq 0$ and has potential to be large.
- In both scenarios, it's still desirable to make the correct margins larger and incorrect margins smaller.

**The SVM cost function**: Let $\theta = \{w_j\}_{j=1,\ldots,c}$ where there are $c$ classes. Then, to optimize $\theta$, we want to minimize the hinge loss across all training examples. Thus, we want to solve the following minimization problem:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \sum_{j \neq y^{(i)}} max[0, 1 + a_j(x^{(i)}) - a_{y^{(i)}}(x^{(i)})]$$

Suppose $1 + a_j(x^{(i)}) - a_{y^{(i)}}(x^{(i)}) = z_j$ and $max[0, z_j] = L_i$. Then we get: $\nabla_{w_j} L_i = \mathbb{1}_{(z_j > 0)} \cdot x^{(i)}$. Also: $\nabla_{w_{y^{(i)}}} L_i = -\sum_{j \neq y^{(i)}} \mathbb{1}_{(z_j > 0)} \cdot x^{(i)}$.

## Optimization

**Stochastic gradient descent**: set a learning rate $\epsilon$ and an initial parameter setting $\theta$. Set a minibatch size of $m$ examples. Until the stopping criterion is met: 1)Sample $m$ examples from the training set. 2)Compute the estimate $g = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta J(\theta)$.3)Make an update $\theta \leftarrow \theta - \epsilon g$. **Momentum**: initialize $v = 0$, and set $\alpha \in [0, 1]$. 1)Compute gradient $g$. 2)Update $v \leftarrow \alpha v - \epsilon g$. 3)Perform gradient step: $\theta \leftarrow \theta + v$. This prevents "zigzag". This modification augments the gradient with the running average of previous gradients. It tends to find "wide"local minimas. **Nesterov Momentum**: $v \leftarrow \alpha v - \epsilon \nabla_\theta J(\theta + \alpha v)$. Then $\theta \leftarrow \theta + v$. **Adaptive gradient**: $\epsilon$ decreases through division by the historical gradient norms. Init $a = 0$, $v = small$. Compute gradient $g$. Update $a \leftarrow a + g \odot g$. Perform step: $\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{a} + v} \odot g$. **RMSProp**: Init $a = 0, v = small, \beta \in (0, 1)$. Update $a \leftarrow \beta a + (1 - \beta) g \odot g$. Perform update: $\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{a} + v} \odot g$. $\beta$ helps "forget"old gradients. **RMSProp with momentum**: Init $a = 0, v = 0, \alpha, \beta \in (0, 1)$. Update $a \leftarrow \beta a + (1 - \beta) g \odot g$. Update momentum: $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{a} + small} \odot g$. Perform update: $\theta \leftarrow \theta + v$. **Adaptive moments (Adam)**: Composed of momentum-like step, followed by Adagrad/RMSProp-like step, with bias correction. Init $v = 0, a = 0, \beta_1, \beta_2 \in (0, 1), t = 0$. Compute gradient $g$. Update time $t \leftarrow t + 1$. Update first moment $v \leftarrow \beta_1 v + (1 - \beta_1) g$. Update second moment: $a \leftarrow \beta_2 a + (1 - \beta_2) g \odot g$. Perform bias correction: $\hat{v} = \frac{1}{1 - \beta_1^t} v, \hat{a} = \frac{1}{1 - \beta_2^t} a$. Perform gradient step: $\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{\hat{a}} + small} \odot \hat{v}$.

## Gradient descent

Recall the definition of the gradient: $\nabla_x f(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{pmatrix}$. To update $x$ so as to minimize $f(x)$, we repeatedly update: $x \leftarrow x - \epsilon \nabla_x f(x)$, where $\epsilon$ called the learning rate (which can change over iterations). Setting $\epsilon$ appropriately is important for convergence. The cost function can be very informative as to how to adjust $\epsilon$. If the cost function diverges fast (as a function of the iteration number), it means that $\epsilon$ is too large - the updates in $x$ are too large. If the cost function converges slowly/plateaus, it means that $\epsilon$ is too small - the updates in $x$ are not significant enough. **Numerical gradient** is too slow to calculate because the dimensionality of $x$ could be large. Therefore, gradient descent is usually faster.
**Batch vs minibatch**: Calculating the gradient exactly is expensive, because it requires evaluating the model on all $m$ examples in the dataset. This leads to an important distinction.

- **Batch algorithm**: uses all $m$ examples in the training set to calculate the gradient.
- **Minibatch algorithm**: approximates the gradient by calculating it using $k$ training examples where $1 \leq k \leq m$.
- **Stochastic algorithm**: approximates the gradient by calculating it over one sample.

It's typical in deep learning to use minibatch gradient descent. Note that some may also use minibatch and stochastic gradient descent interchangeably.
**Robust estimate**: to get a more robust estimate of the gradient, we would use as many data samples as possible: $J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \log p_{model}(x^{(i)}, y^{(i)}; \theta) \Rightarrow \nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log p_{model}(x^{(i)}, y^{(i)}; \theta)$. Thus, when we use more data, we can estimate: $\nabla_\theta J(\theta) \approx \mathbb{E}[\nabla_\theta \log p_{model}(x^{(i)}, y^{(i)}; \theta)]$.

## Neural Networks

**The artificial neuron**:

- The incoming signals, a vector $x \in \mathbb{R}^N$, reflects the output of $N$ neurons that are connected to the current artificial neuron.
- The incoming signals, $x$, are pointwise multiplied by a vector $w \in \mathbb{R}^N$. That is, we calculate $w_i x_i$ for $i = 1, \ldots, N$. We then sum the results and add a bias: $\sum_i w_i x_i + b$.
- The output is then moving through a non-linear transformation $f(\sum_i w_i x_i + b)$, which is called the activation function.

We call the first layer of a neural network the **input layer**. We call the last layer the **output layer**, which we typically represent with the variable $z$ (not $y$, because the output $y$ could be $softmax(z)$ for example). We call intermediate layers the **hidden layers**, which we typically represent with the variable $h$. When we specify that **a network has N layers**, this **doesn't include the input layer**.
The activation function doesn't typically act on the output layer $z$, as $z$ is meant to be interpreted as a vector of scores. A classifier for example may act on $z$, such as $softmax$ or SVM.
**A perspective on feature learning**: one area of machine learning is finding features of the data that are then good for use as the input data to a classifier (like SVM). The intermediate layers of a neural network are features that the later layers then use for decoding. If the performance of the neural network is well, these features are good features. Importantly, these features don't have to be handcrafted.

**Activation functions**
"One recurring theme throughout neural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm."
**Sigmoid unit**: $\sigma(x) = \frac{1}{1+e^{-x}} \Rightarrow \frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$. **Pros**: 1)Around $x = 0$ the unit behaves linearly. 2)The unit is differentiable everywhere. **Cons**: 1)At extremes, the unit saturates and has a gradient which is close to 0. This results in no learning with gradient descent. 2)The unit is not zero-centered; rather, its outputs are centered in 0.5. 3)The gradient will "zigzag"if the inputs to the layer will be all positive: suppose we have $f(w) = \sigma(w^T h + b)$ (which is the output of the layer after the sigmoid activation function) where $h_i \geq 0$ for all $i$ (h is the output of another sigmoid function). Thus, $\frac{\partial f(w)}{\partial w}$ has all positive entries. Now suppose we have an upstream gradient $\frac{\partial \mathcal{L}}{\partial f(w)}$ (which is a scalar since $f(w)$ is a scalar). This means that $\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial f(w)}{\partial w}\frac{\partial \mathcal{L}}{\partial f(w)}$ will have all positive entries or all negative entries, which depends on the sign of $\frac{\partial \mathcal{L}}{\partial f(w)}$.

## Neural Networks

**Hyperbolic Tangent**: $tanh(x) = 2\sigma(x) - 1 \Rightarrow \frac{dtanh(x)}{dx} = 1 - tanh^2(x)$. **Pros**: 1)around $x = 0$, the unit behaves linearly. 2)The unit is differentiable everywhere.3) The unit is zero-centered. **cons** The unit saturates, so for extreme values of $x$ there's no learning.
**ReLU**:$ReLU(x) = max(0, x) \Rightarrow \frac{dReLU(x)}{dx} = \mathbb{1}_{(x>0)}$. **Pros**: 1)In parctice, learning with ReLU converges faster than learning with sigmoid or tanh. 2)When the unit is active, it behaves as a linear unit. 3)The derivative at all points is 0 or 1. When $x > 0$ the gradients are large and not scaled by second order effects. 4)There's no saturation if $x > 0$. **Cons**: 1)The unit is not zero centered. 2)The unit is not differentiable at $x = 0$, but in practice this is not really a problem because we can use the left subgradient (this is reasonable given digital computation is subject to numerical error). 3)Learning doesn't happen for examples that have zero activation. This can be fixed with Leaky ReLU or maxout unit.
**Softplus**: $\zeta(x) = \log(1+e^x) \Rightarrow \frac{d\zeta}{dx} = \sigma(x)$. This unit looks almost the same as ReLU but in addition it's differentiable at $x = 0$. However, ReLU performs better in practice.
**Leaky ReLU**: $f(x) = max(\alpha x, x) \Rightarrow$. The Leaky ReLU avoids the stopping of learning when $x \leq 0$. $\alpha$ may be treated as a selected hyperparameter, or it may be a parameter to be optimized in learning, in which case it's typically called "PReLU"for Paramaterized ReLU.
**ELU (Exponential Linear Unit)**: $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$. The ELU avoids the stopping of learning when $x < 0$. A con of this activation function is that it requires computation of the exponential, which is more expensive.
**Maxout Unit**: $maxout(x) = max(w_1^T x + b_1, w_2^T x + b_2)$. This is a generalization of ReLU and PReLU. This unit doubles the number of parameters, and can even be generalized to more than 2 components. For $w_1 = 0, b_1 = 0$, maxout reduces to ReLU.
**In Practice**: 1)The ReLU unit is very popular. 2)Sigmoid is almost never used (tanh is preferred).
**Which cost function and output function to use?**: The choice of the output function interacts with the cost function to use. Example: suppose we use the sigmoid function as the output of the neural network (thus, there are 2 classes). We can interpret the output as the probability that the input belongs to class 1. Is it better to use Mean Square Error or Cross Entropy as our loss function? $MSE = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - \sigma(z^{(i)}))^2, CE = -\sum_{i=1}^{n}[y^{(i)} \log \sigma(z^{(i)}) + (1 - y^{(i)}) \log\ (1 - \sigma(z^{(i)}))]$. Consider just one example where $y^{(i)} = 1$. We can see that $\frac{\partial MSE}{\partial z^{(i)}}$ saturates to zero when $z^{(i)} \ll 0$ (indicating a large MSE), and so in this case no learning occurs. On the other hand, we can see that $\frac{\partial CE}{\partial z^{(i)}} = \sigma(z^{(i)}) - 1$ is strictly smaller than 0 (the gradient doesn't saturate). Thus, learning will occur for large and negative values of $z^{(i)}$. The learning will only "stall"when $z^{(i)}$ gets close to the right answer (in which case $z^{(i)}$ is large and positive).
**Output activations**:
1)**Linear output units**: $\hat{y} = z$. These output units typically specify the conditional mean of a gaussian distribution: $p(y|z) = \mathcal{N}(z, I)$. In this case, MLE estimation is equivalent to minimizing squared error. 2)**Sigmoid outputs**: typically used for binary classification to approximate a Bernoulli distribution. $Pr(y = 1|x) = max(0, min(1, z))$ won't work in this case because for large or small values of $z$ no learning will occur. 3)**Softmax output**: $\hat{y}_i = softmax_i(z)$. The softmax is the generalization of the sigmoid output to multiple classes.

## Backpropagation

Forward propagation is the act of calculating the values of the hidden and output units of the neural network given an input. Backpropagation is the act of passing the gradients from the output to each hidden unit, all the way to the input layer. **Why do we need backprop?**: 1)Evaluating analytical gradients may be computationally expensive. 2)Backprop is generalizable and is often inexpensive. Note that backprop is not a learning algorithm. It's a method of computing gradients.
**Interpreting backprop as gradient "gates"**: 1)**Add** gate distributes the gradient. 2)**Mult** gate switches the gradient. 3)**Max** gate routes the gradient (to the edge which has a larger value). 4)When two gradient paths converge, we use the rule of total derivatives. Lets say we have $x \to h \to (q_1, q_2)$. Then: $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial q_1}{\partial x}\frac{\partial \mathcal{L}}{\partial q_1} + \frac{\partial q_2}{\partial x}\frac{\partial \mathcal{L}}{\partial q_2}$.
**Multivariate Chain Rule**:
**Gradient w.r.t a vector**: $\nabla_x y = \begin{pmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{pmatrix}$. **Gradient w.r.t a matrix**: $\nabla_A y = \begin{pmatrix} \frac{\partial y}{\partial A_{11}} & \cdots & \frac{\partial y}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial A_{m1}} & \cdots & \frac{\partial y}{\partial A_{mn}} \end{pmatrix}$. In the denominator layout, the dimensions of $A$ and $\nabla_A y$ are the same. **Derivative of a vector w.r.t a vector**: $\nabla_x y = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{pmatrix} \triangleq \frac{\partial y}{\partial x}$. $J$, the Jacobian matrix, is defined as $J(\nabla_x y)^T$. **Derivative of a vector w.r.t a matrix**: if $z \in \mathbb{R}^p$ and $\boldsymbol{W} \in \mathbb{R}^{m \times n}$, then $\nabla_{\boldsymbol{W}} z \in \mathbb{R}^{m \times n \times p}$. Each $m \times n$ slice is the matrix derivative $\nabla_{\boldsymbol{W}} z_i$.

## Backpropagation

**Tensor derivatives**: Suppose we have a vector $z \in \mathbb{R}^p$, a matrix $W \in \mathbb{R}^{m \times n}$, and an input vector $x \in \mathbb{R}^n$ such that $z = f(Wx)$ and suppose we have some scalar loss $\mathcal{L}(z)$. We can calculate $\frac{\partial \mathcal{L}(z)}{\partial W}$ using the chain rule and the total derivatives rule, we get: $\frac{\partial \mathcal{L}(z)}{\partial W} = \frac{\partial z}{\partial W}\frac{\partial \mathcal{L}(z)}{\partial z} = \sum_{j=1}^{p} \frac{\partial z_j}{\partial W}\frac{\partial \mathcal{L}(z)}{\partial z_j}$. So, we can avoid calculating the tensor derivative $\frac{\partial z}{\partial W}$ by calculating $\frac{\partial z_j}{\partial W}$ for each $j$. Remember: $\nabla_x x^T y = y, \nabla_x Wx = W^T, \nabla_W Wx =$ "*out to look like* $x^T$". In general, the simpler rule can be inferred via pattern intuition / looking at the dimensionality of the matrices, and these tensor derivatives need not be explicitly derived. Indeed, actually calculating these tensor derivatives, storing them, and then doing tensor-vector multiply, is usually not a good idea for both memory and computation.

**ReLU layer backpropagation**: let $h = ReLU(Wx + b)$, with $h \in \mathbb{R}^h$, $x \in \mathbb{R}^m$, $W \in \mathbb{R}^{h \times m}$, and $b \in \mathbb{R}^h$. Consider a simple squared loss $\mathcal{L} = (h - z)^2$ where $z$ is some target value. Lets mark $a = Wx + b$, $c = Wx$: $\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial h}{\partial a}\frac{\partial \mathcal{L}}{\partial h} = \mathbb{1}_{(a>0)} \odot \frac{\partial \mathcal{L}}{\partial h} = \frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial c}$. And most importantly: $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial c}{\partial x}\frac{\partial \mathcal{L}}{\partial c} = W^T\frac{\partial \mathcal{L}}{\partial c}$, $\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial c}{\partial W}\frac{\partial \mathcal{L}}{\partial c} = \frac{\partial \mathcal{L}}{\partial c}x^T$. **In the denominator layout, the chain rule is written from RIGHT TO LEFT**.

## Regularization

**Small random weight initialization**: Empirically, these initializations cause all activations to decay to zero, especially in deeper hidden layers. In addition, backpropagation gradients will be very small and cause no learning. In practice, small weight initializations may be appropriate for small neural networks. **Large random weight initialization**: cause the units to explode. Gradients also explode. Thus, we need an intermediate initialization.

**Xavier Initialization**:

"The variance of the units across all layers should be the same, and that the same holds true for backpropagated gradients": $var(h_1) = \ldots = var(h_n)$, $var(\frac{\partial \mathcal{L}}{\partial h_1}) = \ldots = var(\frac{\partial \mathcal{L}}{\partial h_n})$. If the units are linear, then $h_i = \sum_{j=1}^{n_{in}} w_{ij}h_{i-1,j} =$. Further, if $w_{ij}$ and $h_{i-1}$ are independent, and all the units in the $(i-1)th$ layer have the same statistics, then assuming $E[w] = E[h] = 0$, we get: $var(w_i h_{i-1}) = \mathbb{E}^2[w]var(h) + \mathbb{E}^2[h]var(w) + var(w)var(h) = var(w)var(h) \Rightarrow var(h_i) = var(h_{i-1})\sum_{j=1}^{n_{in}} var(w_{ij})$. We force that $var(h_i) = var(h_{i-1})$, thus: $\sum_{j=1}^{n_{in}} var(w_{ij}) = 1$. The weights should be identically distributed, and so: $n_{in}var(w_{ij}) = 1 \Rightarrow \boldsymbol{var(w_{ij}) = \frac{1}{n_{in}}}$, where $n_{in}$ is the number of units in the prior layer. Also, the same argument could be made for the backpropagated gradients: $var(w_{ij}) = \frac{1}{n_{out}}$. To incorporate both of these constraints, we can average the number of units together, to get: $\boxed{var(w_{ij}) = \frac{2}{n_{in} + n_{out}}}$. We can then initialize each weight in layer $i$ to be drawn from $\mathcal{N}(0, \frac{2}{n_{in}+n_{out}})$. **Note**: Xavier initialization usually leads to dying ReLU units, though it's fine with tanh units. To fix this, we can set $var(w_{ij}) = \frac{2}{n_{in}}$: If linear activations prior to ReLU are equally likely to be positive or negative, ReLU kills half of the units, and so the variance decreases by half. This motivates an additional factor of 2. **A factor of 2 in the initialization can be the difference between a network that learns and a network that doesn't**.

**Batch normalization**:

An obstacle to standard learning is that the distribution of the inputs in each layer changes as learning occurs in previous layers. As a result, the unit activations can be vary a lot. Another consideration is that when we do gradient descent, we are calculating how to update each parameter **assuming** the other layers don't change, but these layers may change drastically. These cause: 1)Learning rates to be smaller. 2)Networks to be more sensitive to initializations. 3)Difficulty in training networks that saturate. **Batch normalization** makes the output of each layer with unit statistics. Learning then becomes simpler because parameters in the layers don't change the statistics of the input to a given layer. This makes learning more straightforward. In layer $j$ for example $i$: $\hat{x}_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$, $\mu_j = \frac{1}{m}\sum_{k=1}^{m} x_j^{(k)}$, $\sigma_j^2 = \frac{1}{m}\sum_{k=1}^{m}(x_j^{(k)} - \mu_j)^2$, where $\epsilon$ is small. Then, after scaling the inputs of the batchnorm layer $j$, we scale and shift the output of it: $y_j = \gamma_j \hat{x}_j + \beta_j$. The normalization and scale/shift operations are included in the computational graph of the neural network, so that they participate not only in forward propagation, but also in backpropagation. **Notes**: 1)The reason the scaling is on per unit basis is primarily computational efficiency. It's possible to normalize the entire layer via $\Sigma^{-\frac{1}{2}}(x - \mu)$. However, this requires computation of a covariance matrix, its inverse, and the appropriate terms for backpropagation (including the Jacobian). 2)The scale and shift layer is inserted in case it's better that the activations won't be zero mean and unit variance. As $\gamma_j$ and $\beta_j$ are parameters, it's possible for the network to rescale the activations. In fact, it can undo the normalization. 3)A batch-norm layer is typically placed right before the nonlinear activation.

**WRITE BATCHNORM BACKPROP**

## Regularization

**Regularization** is any modification we make to a learning algorithm that is intended to reduce its generalization but not its training error. It can be seen as a way to prevent overfitting. It may help make the model less complex (large model that is regularized properly).**Types of regularization**: 1)Adding a soft constraint on the parameter values in the objective function. 2)Dataset augmentation. 3)Ensemble methods. 4)Some training algorithms - stopping early, dropout, etc.

**Regularization methods**: 1)Stopping early: requires caching the lowest validation error model. Training will stop after a pre-defined number of iterations. 2)Parameter norm penalties: $loss = \mathcal{L}(\theta; X, y) + \alpha\Omega(\theta),\ \alpha \geq 0$. $L2$ norm promotes models with parameters close to 0. Using prior knowledge of $w$ can also be done by minimizing the $L2$ between $w$ and the prior knowledge of $w$. We can also minimize the L2 loss between $w_i$ and $w_j$ if we know that 2 weights should be close to each other. $L1$ regularization results in sparse solutions. $L1$ can also be used for feature selection. 3)Transfer learning: retrain only the last layers. 4)Ensemble methods: train multiple different models, average their results together at test time. This almost always increases performance.

**Ensemble methods**: If models are independent, they usually not all make the same errors on test set. With $k$ independent models, the average model error will decrease by a factor of $\frac{1}{k}$.

**Bagging**: construct $k$ datasets using bootstrap (set a dataset size $N$, and draw with replacement from the original dataset to get $N$ samples. Do this $k$ times). Then, train $k$ different models using these $k$ datasets.

**Dropout**: On a given training iteration, sample a binary mask for all inputs and hidden units in the network. Apply the mask to all units, and then perform a forward pass and a backward pass, and parameter update. In prediction, multiply each hidden unit by the parameter of its Bernoulli mask, $p$. **Notes**: 1)an advantage of dropout over ensemble methods, is that on test time there is not additional complexity. 2)Dropout approximates bagging. 3)Dropout regularizing each hidden unit to work in many different contexts. 4)Dropout may cause important features to be encoded in many different neurons. **Inverted dropout**: a common way to implement dropout is inverted dropout where the scaling by $\frac{1}{p}$ is done in training. This causes the activations to have the same expected value as if dropout has never been performed. Thus, the testing looks the same irrespective of if we use dropout or not.

## Convolutional Neural Networks

CNN's actually don't do convolution. They calculate correlation: $(f * g)(i, j) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m, n)g(i + m, j + n)$. Note that correlation is not commutative like regular convolution. All convolutions in this class are "valid"(we explicitly specify padding).

**Basic operation of a convolutional layer**: Suppose the input dimensions are $H_{in} \times W_{in} \times D_1$ and that we have $D_2$ filters of dimensions $H_{filter} \times W_{filter} \times D_1$. Then: $W_{out} = W_{in} - W_{filter} + 1$, $H_{out} = H_{in} - H_{filter} + 1$, and so the output dimensions would be $H_{out} \times W_{out} \times D_2$ (each filter is operating across the whole $D_1$ dimension). The output is then passed through an activation function, and then this acts as the input to the next layer.

Convolutional layers have sparse connectivity. Each output is connected to a small number of inputs. This reduces computational memory and time. If there are $m$ inputs and $n$ outputs in a hidden layer, a fully connected layer would require $O(nm)$ operations to compute the output, whereas a convolution layer would require $O(kn)$ where $k$ is the number of inputs each output is connected to.

**Why more layers?**: Because of sparse connections, information from different parts of the inputs may not interact. This argues that networks should be composed of more layers, since units in deeper layers indirectly interact with larger portions of the input. **Conv layers share parameters**: every output neuron in a given slice uses the same set of parameters in the filter of that slice. **Conv layers can handle variable size inputs**: the output size may be different, but the input size doesn't matter, whereas in a fully connected network it does.

**Padding**: Valid convolutions decrease the output size. The input and output $W$ dimension will be equal if the input is padded with $\frac{w_{filter}-1}{2}$ zeros in the $W$ dimension, in each side (same goes for the $H$ dimension). When we specify a variable *pad*, it's the amount of zeros to pad on each border. **With padding, the output size is now** $(H_{in} - H_{filter} + 1 + 2pad, W_{in} - W_{filter} + 1 + 2pad, D_2)$.

**Stride**: stride defines how much the filter moves in the convolution. For normal convolution, stride=1. With stride, the output size would now be

$$\boxed{(\frac{H_{in} - H_{filter} + 2pad}{stride} + 1, \frac{W_{in} - W_{filter} + 2pad}{stride} + 1, D_2)}$$

**Pooling**: Pooling is a form of downsampling. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input. The pooling layer (filter) has and height and width $(H_p, W_p)$ and is applied with a given stride. It's most common to use the max pooling. The output dimensions would be $(\frac{H_{in}-H_p}{stride} + 1, \frac{W_{in}-W_p}{stride} + 1, D_1)$. Note that $D_1$ stays the same as the input since pooling operates on each slice independently